1994

Problem Solving Environments for Partial Differential Equation Based Applications (Ph.D. Thesis)

Sanjiva Weerawarana

Report Number:
94-058

PROBLEM SOLVING ENVIRONMENTS FOR PARTIAL DIFFERENTIAL EQUATION BASED APPLICATIONS

Sanjiva Weerawarana

CSD-TR-94-058
August 1994
(Ph.D. Thesis)

Problem Solving Environments for Partial Differential Equation Based Applications

Sanjiva Weerawarana
Ph.D. Thesis

August 1994

Dedication

This thesis is dedicated to my parents, Kamal and Ruby Fernando, without whose lifelong advice, guidance, encouragement, trust, patience and unbounded love my academic career could never have reached this point.

Acknowledgements

It is a pleasure to take this opportunity to attempt to recognize all the people who helped me get through my (sometimes difficult) doctoral student years!

Since I can only name so many people, I will start by saying "Thank You!" to all my wonderful friends, teachers and of course relatives. I am very sorry that I cannot acknowledge you individually, but the appreciation I have for what you all did for me is nonetheless limitless!

Of course, my wonderful advisor Prof. Elias N. Houstis is the catalyst to almost every good thing that happened to me in the last several years. A short paragraph cannot possibly acknowledge all that he has done for me; Elias Houstis was much more than an academic advisor to me in these years. To Elias, I owe eternal gratitude, infinite respect and lifelong friendship for (extremely patiently) moulding a researcher out of me!

Prof. John R. Rice also provided much inspiration during my time in the CAPO, SoftLab and PDELab groups. His careful scrutinization of my thesis helped it immensely and the many discussions with him taught me to not always worry about details! The other two members of my examining committee, Prof. Vincent F. Russo and Prof. Ahmed K. Elmagarmid, also helped me in many ways with the suggestions and comments about my work.

Being a part of the CAPO, SoftLab and PDELab groups has been an absolute pleasure for me. Ann Christine, Shahani, Amy, Sang–Bae, Cheryl, Stephen, Po Ting, Kostas, Nick (Houstis), Tzvetan, Anupam, Ben, Margaret and Mei–Ling and past members, Panos, Ko Yang, Nick (Chrisochoides), Reggie, Yu–Ling, Manolis, Pelayia, Jim, Scott and Hyeran have all contributed to my work with all the discussions we've had about how to solve PDEs. Ann Christine needs special mention; thanks so much for all your invaluable help, advice and encouragement, Ann Christine!

If I didn't acknowledge the group of Sri Lankan friends with whom I spent many an evening, then I'd be doing a great injustice. Our late night sessions coming up with solutions to all the problems of the world (after a few rounds, of course) and the volatile walleyball sessions were probably the most therapeutic things I did during these years! To Yohan, Nishantha, Kirthi, my little sister Santhani, Niranjan, Athula, Rajee and of course my wife Shahani, "Thank you!", I will never forget these great years!

Now to the people closest to me– of course, none of this would ever have been possible had my family not allowed me to come to the US in 1985. I do not have the right words to acknowledge my father Kamal, my mother Ruby, my elder sister Kumari, my younger sister Santhani and my brother–in–law Varuna, so I will simply say "Thank you!" Your support never waivered (even when I was terribly waivering ...) and you never stopped telling me to keep going – I know I couldn't have done it without your help!

Finally, and obviously the most, I come to my beloved wife, Shahani Markus. Even though we've been married only for a bit over two years, we've known each other since just two hours after I drove over to Purdue in 1989. Shahani has been

a big part of my life and my Ph.D. for much longer than two years and has had to tolerate all my bickering about them. For all the patience, kindness, advice and love you have given me, I thank you. For putting up with all the stuff that I've been throwing at you (not literally, of course!) for the last several years, I thank you. For staying up with me the night before my defense and teaching me those wonderful German things so that I could pass the language exam the next morning, I thank you (big time)! As I cannot possibly explain all the little things that you did for me, I will just say "I simply couldn't have done it without you, period!"

Really finally, I wish to acknowledge all the generous funding agencies that made my work logistically possible: Purdue University, Purdue University Foundation, Purdue University Research Foundation, Intel Foundation, National Science Foundation, Air Force Office of Scientific Research, Advanced Research Projects Agency, and my parents' bank account.

Contents

List of Tables

List of Figures

Abstract

Problem solving environments represent the next generation of scientific computing software. This thesis deals with four aspects of building problem solving environments for partial differential equation based applications: software architecture of development frameworks, high level languages, computational intelligence, and integrating experimentation and computation.

The software infrastructure available for building problem solving environments is low level and is designed for computer scientists rather than computational scientists. A software framework that allows computational scientists to develop their own problem solving environments for partial differential equation based applications is presented. The framework is a multi–layered architecture that provides infrastructure at all levels of the problem solving environment development process.

High level, formal notations are a useful vehicle for expressing partial differential equation problems and their solution schemes. A high level symbolic language that supports the specification of arbitrary differential models and includes powerful solution scheme specification capabilities is presented. The language is based on decomposing the problem into its constituent parts and on decomposing the solution process into a sequence of transformations.

Along with the increased complexity of scientific computing software, the choices and decisions users must make have also become increasingly complex. A technique for intelligently selecting optimal partial differential equation solvers is presented. The reasoning methodology uses the exemplar reasoning approach and is based on the observed performance of various solvers on a set of existing problems.

Finally, the problem of integrating experimentation and computation is addressed. The idea of a "virtual science laboratory" that integrates experimentation and computation in one cohesive environment is developed. Methodologies for using the integrated environment in several scenarios are also presented. The environment is based on the problem solving environment infrastructure developed in this thesis and includes laboratory instrument control and data acquisition facilities.

Chapter 1

Introduction

Ever since the early days of computing, the use of computers to solve scientific or physical problems has been a driving force in the effort to advance the field of computing. As early as the 1960s, one can find attempts at developing software platforms to solve such problems ([CF63, KR68]). These early efforts mostly failed because of the lack of computing power and the limitations of early programming languages.

While scientific applications still drive the need for more and more powerful hardware, they also demand the development of more and more powerful software systems. Scientific computing as practiced today, ranges from writing special–case solvers in assembly languages and FORTRAN–type languages to applying advanced software systems with multimedia interfaces that quickly compute solutions using parallel machines on a network and provide virtual reality visualization to help understand the phenomena under study. This thesis studies the problem of developing such advanced software platforms for scientific applications whose mathematical models are based on partial differential equations (PDEs).

The rest of this chapter introduces the reader to PDEs and how they are solved, to software for solving PDEs, to the needs of PDE–based applications and to some issues that arise when developing software for such applications. Finally, we introduce a prototype environment developed as a part of this research and then provide an outline of the rest of this thesis.

1.1 Modeling with Partial Differential Equations

Partial differential equations are mathematical equations that involve some number of unknown quantities and their derivatives expressed in terms of some number of known quantities. Let $\vec{u} = (u_1, u_2, \ldots, u_k)$ be the vector of unknown variables and $\vec{x} = (x_1, x_2, \ldots, x_n)$ be the vector of known variables. Then,

$$F(\vec{x}, \vec{u}, \vec{u}_{x_1,\ldots,x_m}, \ldots, \vec{u}_{x_n,\ldots,x_k}) = 0$$

is called a *partial differential equation* and F is called a *partial differential operator*, where $\vec{u}_{x_i}$ is the (partial) derivative of $\vec{u}$ with respect to x_i. The *domain* of the

problem is a subspace of $\mathcal{R}^n$ and is where the equation is said to hold. At the boundary of the domain, a different set of equations, called *boundary conditions* hold. If the phenomenon is transient, then another set of equations, called *initial conditions* describe the initial state of the system.

Most physical phenomena can be naturally expressed in terms of the condition of the phenomena at a certain time and by how it changes with respect to time. These changes with respect to time are often expressed in terms of the historical condition and by the effect of any internal/external forces that are present at the current time. The terms "phenomena", "condition", "time", "force", and "change" all have direct counterparts in the world of partial differential equations. Hence, PDEs are a commonly used mathematical tool for modeling physical phenomena.

Although partial differential equations are convenient modeling tools, they are not generally solvable analytically. That is, only certain classes of (mostly simple) PDEs can be solved exactly. Hence, a large number of numerical solution techniques that compute an approximate solution have been developed over the years. While general systems of partial differential equations are interesting and do occur in models of physical phenomena, high order systems are generally not used for numerical simulations. Since higher order systems of PDEs can easily be transformed to systems of lower order equations via an appropriate change of variables, this is not a difficulty. Hence, we only concern ourselves with systems of nonlinear, time–dependent second order partial differential equations. Furthermore, since we are concerned with modeling physical phenomena, we restrict ourselves to 1–, 2– and 3–dimensional domains.

Solving partial differential equations numerically is a vast area with hundreds and probably thousands of techniques in regular use. Of these, finite differencing, finite element, Monte–Carlo, spectral and variational techniques are the most important because of their generality. Since the software issues and environments we consider in this thesis deal primarily with finite difference and finite element techniques (the most widely used techniques), we will provide very brief introductions to these techniques.

Finite difference techniques compute the solution and possibly its derivatives at points distributed in some way over a grid in the domain. The basic idea is to replace the (continuous) derivatives in the PDE operator by *difference quotients* (approximations to the derivatives at a point using values at nearby points) and then solve the resulting system of algebraic equations. Finite element techniques compute the solution on small regions of the domain ('elements') by expanding the unknown functions in terms of a set of basis functions. Solving the resulting system of algebraic equations gives the coefficients of the expansion. In either method, nonlinearity can be dealt with either at the PDE operator level or at the algebraic equation level via some appropriate linearization technique. In any case, it eventually becomes necessary to solve a large system of linear equations.

Software for solving specific classes of PDE problems started appearing as early as the late 1960s. These early solvers handled relatively small classes of problems and consisted of a library of routines that the user could apply to obtain solutions. These systems have evolved over the years; but as we will see in the next section, low level

packages still abound.

1.2 Evolution of PDE Solving Software

A large group of people who use partial differential equation models often deal with one or a few different models throughout their 'modeling' lifetime. Consider, for example, a civil engineer who designs bridges and is therefore interested in the structural properties of various bridge designs and various materials that can be used for a particular bridge. Using specific information about the bridge under consideration (such as the material properties of the components, the loads the bridge should be able to withstand and the design of the bridge itself), the engineer would need to solve the PDE model of the structural analysis problem. Once the solution is available, the engineer may use the computed stress and displacement information to optimize the bridge design and finally generate a set of specifications that are sent to the builders.

For these types of "standard" PDE models, there are commercially available software packages (for example, [Swa85, Mac91a, Hib89]) that have very good built–in solvers. They also include some "pre–processing" component that helps the engineer specify the material properties, etc. and a "post–processing" environment where the engineer can visualize solutions in terms of the effect on the particular design (s)he is working with. These commercial packages can generally solve one or a small number of models; the user simply selects the model, specifies the relevant parameters and lets the package do the rest.

On the other hand, a large group of people who use partial differential equation models deal with application–specific, problem–specific models. For example, in one of the benchmark applications we will describe later in this chapter, the problem is modeled by a system of two PDEs; one 1–dimensional PDE and one 2–dimensional PDE where the PDEs themselves do not have any standard form (unlike the structural analysis model). For these models, there are *no* commercially available packages that provide even a fraction of the convenience provided by the engineering–type packages described above. Our work attempts to improve the lot of users of such models.

In the rest of this section, we will briefly trace the evolution of PDE solving software from libraries of solvers to intelligent, interactive software systems. However, as we will see, although the technology has evolved, most PDE solver developers and users still work at the library level, mainly because of the restrictions placed by the various evolutionary stages. Figure 1.1 provides a graphical view of the evolution described below.

Earliest to appear were library based PDE solving systems. In terms of architecture, these systems provided a library of solvers and the user would write a driver program that described the problem to be solved in the form expected by the library and then call the solvers. Most of these systems are implemented in FORTRAN.

The next major evolution came with the development of high level application–specific language based systems. These systems' back–end was still a library of solvers, but the user was provided a high level language with which to express the PDE prob-

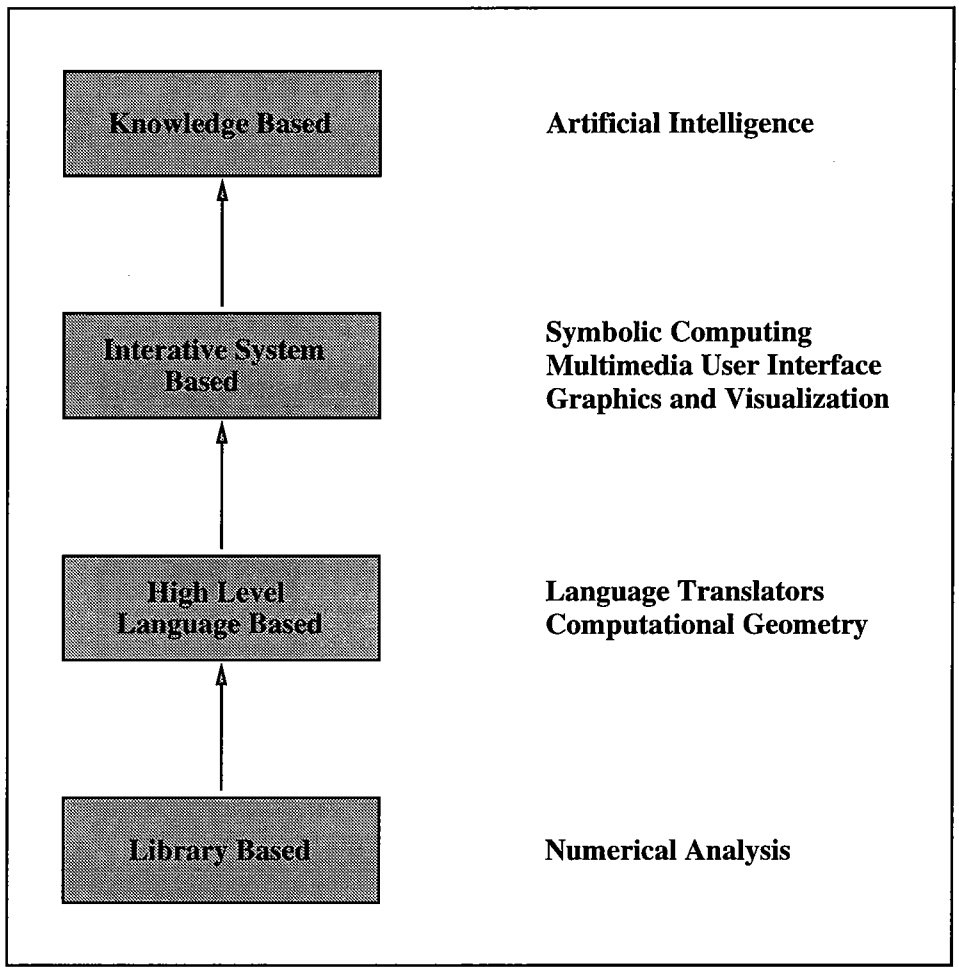

Figure 1.1: Evolution of PDE solving software. The left column provides a characterization while the right column provides a broad listing of the technologies included at each level (higher levels include the technologies used at the lower level).

Table 1.1: Classification of existing PDE solving systems.

Class	Examples
Library Based	VECFEM, FIDISOL, CADSOL, PDECOL, PDEONE, PDETWO, KASKADE
High Level Language Based	ELLPACK, PDEQSOL, PDE/PROTRAN
Interactive System Based	//ELLPACK, Visual PDEQSOL, ALPAL
Knowledge Based	EVE, Elliptic Expert

lem rather than forcing them to express it (rather unnaturally) using FORTRAN. A language processor then translated this specification into an automatically generated driver program which called on existing libraries to solve the problem. High level interchangeable solver parts also evolved during this stage; the user was provided with the option to specify the solution strategy as a composition of several parts with many options for each part.

Interactive systems evolved PDE solving software by providing users with a (graphical) programming environment for developing the high level language program. This pre–processing environment was also complemented with a graphical post–processing environment which provided visualization and analysis capabilities. There was generally a loose coupling between this programming and analysis environment and the solution stage itself. Most currently existing commercial PDE solving software have such architectures, except that an explicit high–level language is not used.

The next evolution came with applying artificial intelligence to assist users to solve the PDE problem. Knowledge–based systems generally used some kind of domain knowledge to advice users on how to solve a specific PDE problem. In some cases, the systems are directly and tightly coupled to the numerical solvers. While this generally provides more latitude for reasoning than in other cases (as detail understanding of the numerical methods are then available), solution capabilities of such systems are limited due to the difficultly (in terms of human cost) of performing this tight coupling (e.g., EVE [BBP+92]). In other cases, the knowledge–based analysis behaves as a preprocessor to an interactive or language based system (e.g., Elliptic Expert [DG89]).

Table 1.1 lists some example existing systems that belong to each category described above.

1.3 Problem Solving Environments

Problem solving environments are software environments that assist in the development, solution and analysis of some problem. Restricting to PDE based problems, in [Mac91b] MacRae decomposes the problem solution to the stages of 1) formulating ideas into mathematical representation, 2) describing variables and relations, 3)

setting up equations and selecting approximations, 4) selecting (PDE) models, 5) optimizing models using experimentation, 6) solving the models, and 7) interpreting results and modifying the models if necessary. A problem solving environment (PSE) is then a software environment that provides support for this entire process. That is, a PSE is a software environment that epitomizes Hamming's quote: "The purpose of computing is insight, not numbers." [Ham62].

1.3.1 Properties of PSEs

A key feature of PSEs is their ability to converse with the scientist in his/her own terms. For example, a PSE assisting a civil engineer must understand what "stress" means and should use that term when it communicates with the scientist. Conversing does not only mean understanding what the scientist requests, but also that the system must be aware of the basics of the field. If one views computer algebra systems as mathematical PSEs, then they exhibit good abilities of conversing in mathematical terms.

A modern computational scientist attempting to use a software library for a simple operation such as matrix factorization would be overwhelmed by the choices that are available. Depending on the properties of the matrix, he/she would need to use a particular routine that best applies to that matrix. Of course, it is always possible to use a general method that works for all matrices; but, this amounts to a waste of computing resources that could easily have been avoided if the software could automatically select the right method to use. This is another important feature of PSEs; they must *intelligently* assist the user with the selection of state–of–the–art methods for solving certain problems.

It is not sufficient for a PSE to address only one dimension of the problem at hand. For example, in the parallel solution of PDEs, the PSE must provide tools for automatically selecting an appropriate machine from the set of available parallel processors, map the problem on to the selected machine, gather the solution and, finally, assist in performance evaluation.

Another important property of PSEs is their behavior as the manager of computing resources related to the problem. For example, for the parallel solution of a problem, it is important that the PSE manages issues such as machine allocation, mapping, and deallocation. Distribution of tasks within a network of computing agents is another task that the PSE must manage. If the environment has multimedia capability, then the PSE must manage these aspects as well in order to provide a comprehensive and cohesive user interface that exploits all available media in visualizing the problem and its solution.

An example of an excellent PSE exists today in everyone's world. Consider the problem of balancing a check book. The PSE used to solve this problem consists of a notebook and a calculator. These two components interact with each other to solve the problem effectively and completely. Our vision for PSEs for scientific computing

problems is towards this notebook–calculator paradigm. We envision an intelligent electronic notebook along with a set of "calculators" to collaborate in assisting the user solve the problem.

1.3.2 PSEs vs. PSE Frameworks

One cannot fail to notice that in the broad categories of properties of PSEs listed above, there would be many properties common to a set of problems of a certain class. For example, PSEs for PDE–based applications would all have the need for software that assists in the selection of methods for solving PDE problems. Similarly, almost every scientific computing system needs the capability to visualize one, two, three and even four and five dimensional data sets.

We therefore define a *PSE framework* as the basic structure or skeleton upon which PSEs are built. This skeleton is a generic tool that can be used in developing PSEs for a broad class of applications. Rather than attempt to identify and build the framework for all PSEs, we restrict ourselves to the task of building a framework for applications that are based on PDE models.

Our eventual goal is a view towards a level beyond PSE frameworks: the computer science aspects of PSEs should be synthesized into a PSE generating environment. Such a PSE generator would basically consist of a database of PSE "components", navigation tools to search through the database and a (visual) PSE synthesizer language to connect together several components to form a PSE. The Explorer[Sil92] software from Silicon Graphics, Inc. can be viewed as an example of a PSE generator environment for generating PSEs for the problem of scientific visualization.

1.4 PDE Based Applications and Application PSEs

Our goal is to improve the software infrastructure available for building PSEs for PDE based applications. However, as we do not address the needs of *all* PDE based applications, let us clarify the types of applications our work is aimed at.

As mentioned earlier, there exist commercial, mature software systems for solving "standard" PDE models such as stress–strain problems. However, for non–standard PDE models, the existing software base is at best a collection of libraries of solvers that the application scientist must attempt to compose to achieve the desired solutions.

In a typical PDE based application, the PDE models are fairly complex and almost always nonlinear and transient. The PDE model of the problem is generally developed by hand and includes parameters that are determined experimentally. If a solver is already available, then almost always it is a custom developed solver implemented as one large FORTRAN system. User interfaces are also custom built from primitive tools as are visualization tools.

While the technology of PDE solving software has evolved over the years, the base technology used by application scientists for building application specific PDE solvers are still, at most, library based systems. The primary reason for this is that PDE

solving software has been centered on the PDE solution process whereas users are interested in their entire application, not just the solution of a PDE problem. Existing higher–level PDE solving systems assume that human users are their primary users, not other application software. Hence, when building an application specific code, it is not possible to use these systems directly as they are not designed to work with other software.

Our research develops software and algorithmic infrastructure towards the development of a science base that will significantly ease the process of implementing a particular analysis code. This requires the development of a very high level software framework that can be used to develop application specific analysis codes. To be a successful framework, software infrastructure is needed in many areas of PDE solving software including geometry handling, symbolic manipulation, user interfaces, numerical solvers and parallel computing.

1.4.1 The PDELab Prototype

As a part of our research, we have developed a prototype environment called PDELab. This environment has two main functions. The first, of course, is to be a a software framework for developing PSEs for PDE based applications. Secondly, PDELab is a framework for developing PDE solving software and integrating them into one infrastructure. The latter functionality is essential in order to be effective at the first; the system must have comprehensive PDE solving facilities and that goal cannot be achieved unless a methodology for integrating existing software and support for developing new software is available.

The PDELab architecture is based on decomposing the PDE problem and solution process to a set of "PDE objects" that arise naturally. It also includes a high level symbolic language that allows the specification of arbitrary PDE models and complex solution schemes. Furthermore, a framework for using intelligent reasoning for selecting solution schemes and a generic methodology for integrating existing software and supporting the development of new PDE software is developed.

1.5 Overview of the Thesis

This chapter has provided a broad introduction to the issues addressed in this thesis. In Chapter 2 we discuss the design of the PDELab framework. Chapter 3 discusses the high level symbolic PDE specification language we developed. Chapter 4 discusses PYTHIA, an intelligent reasoning framework for selecting "optimal" PDE solvers. Chapter 5 discusses the development of problem of building PSEs that truly integrate computation and experimentation. Finally, Chapter 6 summarizes the thesis and highlights our future work plans.

Chapter 2

The Architecture of a Software Framework for Building Problem Solving Environments for PDE Based Applications

In this chapter we present an object–oriented methodology and tools for creating high level, high performance problem solving systems (workbenches) for scientific applications modeled by partial differential equations. This methodology is validated by the creation of a scientific computing workbench for bioseparation analysis. One of the design objectives of PDELab is to provide workbench developers and users with much the same kind of independence in software as they have come to expect in hardware. The architecture of this software platform for creating problem solving environments for PDE applications is based on "clean layering." At the bottom are the various "smart" libraries that support the numerical simulation of various "physical" objects together with the corresponding knowledge bases needed to support the computational intelligence aspects of the various workbenches; at the top is a set of interactive tools that allow the user to carry out his objectives using "natural" tools. Between these layers sits a piece of "middleware" called a "software bus." Its design objective is to allow the integration of a variety of software components needed to support "hybrid" (numeric and experimental) PDE based workbenches. Moreover, it comes with a software tool that allow its reconfiguration for specific applications. This chapter discusses the design and implementation issues of this three layered architecture of PDELab.

2.1 Introduction

After an application scientist derives a PDE model for a physical process, they need to acquire numerical solution software to solve the model. While there are highly developed software packages for solving commonly arising PDE models such as stress/strain

models, there exists only a very low level software base for solving non–standard models such as the PDE model of bioseparation (see Section 2.9). Application scientists working on such esoteric models must hence develop numerical solution code to solve their PDE model by starting with low level libraries (including linear algebra libraries, ordinary differential equation (ODE) solvers and some PDE solvers), texts and research papers that explain to them the intricacies of various methods and the when and how of combining them to obtain a useful solution. Once a solution has been found, they typically develop their own analysis and visualization code as they believe that their need is quite unique or because available packages are too complex for their needs.

There are many problems with this approach, of course. Firstly, application scientists need to spend significant effort (both in terms of time and mental energy) on learning and developing numerical solution techniques that by itself leads to very little benefit in their own field; that is, scientists spend time practicing computing instead of practicing science. Secondly, since most application scientists are not trained computer scientists, the code they develop is rarely developed in an extensible, reusable manner; thus, for similar analyses, they repeat the entire process rather than possibly adapting some existing code. Furthermore, because of ad hoc development and maintenance methodology, the code often "stops working" once the original code developer leaves. What is needed to improve the application scientist's lot is a science base that reduces both the time and effort to construct, use and maintain such analysis code.

To define a particular analysis code, one chooses equations that best capture the phenomena to be studied, selects appropriate solvers for the equations (accounting for the necessary trade–offs between speed, accuracy, numerical robustness and so on) and defines control mechanisms in the case of sophisticated simulation codes. In addition, one defines appropriate object representations, constructs user– and inter–analysis interfaces that comprise a convenient computational environment and, finally, considers the particulars of the hardware platforms used.

The actual analysis code is of course only a small part of the larger process that the scientist follows. To develop the mathematical model of the phenomena under study, one typically uses symbolic mathematical derivations. Then, the model is transformed into a form that is understood by the solution environment and then solved. Once a solution has been computed, the scientist visually and numerically analyzes the computed data and possibly improve the model or the transformed model and repeat the entire process. This stage may also involve performing actual experiments and comparing computed data with the experimentally obtained data.

The objective of this work described in this chapter is to design a framework for building software environments that provide all the computational facilities needed to solve target classes of problems "quickly", by communicating in the user's terms. In this work we focus on PSEs for scientific applications where the underlying phenomena are modeled by partial differential equations. In general, the PSE technology is expected to reduce the time between an idea and validation of the discovery, to get a "quick" answer to almost any question that has a readily computed answer, support

programming–in–the–large, provide "knowbots" (intelligent agents) that implement various scientific problem solving processes, and allow easy prototyping [GHR92]. We describe the design of a software platform (PDELab) for the development of PSEs for PDE based applications that realize to a degree some of the above expectations. PDELab consists of three layers. The bottom layer involves the various meta–libraries (their modules consist of code and knowledge related to their computational behavior) for the numerical simulation of various physical objects and knowledge bases that support the computational intelligence of the domain specific workbenches. Version of these libraries are assumed to be available on specialized, generic and virtual machines. At the top layer a set of "natural" tools are provided that allow the user to specify the input and to interact and observe the various facets of "hybrid" (numerical and experimental) models used for the simulation of various applications. These tools allow the user to solve problems by communicating in the user's terms. This layer is currently implemented using X11 Window System based technologies. Between these two layers sits a piece of "middleware" called a "software bus" whose design objective is to hook up a range of independent subsystems and tools. For the development of customized workbenches the software bus can be customized with an attached reconfiguration tool.

This chapter is organized as follows. In Section 2.2 we briefly review some related work in the areas of PDE solving software environments and problem solving environment development frameworks. Section 2.3 presents the design objectives of PDELab. Section 2.4 discusses the software architecture of PDELab. Section 2.5 gives a detailed view of the PDELab software bus. In Section 2.6 we describe the application workbench development environment. In Section 2.7 we highlight the integration of symbolic computation in PDELab. Then, we provide two examples to illustrate PDELab's capabilities: Section 2.8 describes the process of integrating a new PDE solver and Section 2.9 describes the application of PDELab to the analysis of bioseparation.

2.2 Related Work

Software environments for scientific computing is a very active, vast area. Others' work in several areas are relevant to our work; hence, we consider related work in the ares of PDE solving software environments, application development frameworks and software bus environments. The latter two are, of course, not specific to scientific computing.

2.2.1 PDE Solving Software Environments

As outlined in the introduction, software environments for solving PDEs have evolved over the years to complex, multilingual systems. Historically, one can see a bifurcation of these systems based on the intended scope– specific model oriented systems for common mathematical models are commercially available while model independent,

generic, library–driven systems have mostly been products of the academic world. As model specific packages do not apply in the topic of this chapter, we will briefly review only the generic systems here.

The most flexibility in terms of being adaptable to developing specific application environments is, of course, provided by the low–level library type systems. Examples of these systems include VECFEM [GS91], PDECOL [MS79], PDETWO [MS81], FIDISOL [SSM85] and CADSOL [WSS92]. The price for flexibility is that one must develop all the related software necessary to build an application problem solving environment; not a trivial task by any measure.

High level language based systems (for example, ELLPACK [RB85], DEQSOL [Hit90] and PDE/PROTRAN [IMS86]) and their recent interactive system counterparts, //ELLPACK [HRC+90] and Visual DEQSOL [UKO92], provide significantly more functionality for the application developer. However, all these systems suffer from the phenomenon that they were not designed to be a part of a larger application; they were designed as systems to solve PDE problems, not as systems to build software environments for applications based on partial differential equations. As a representative of this generation of software, we will consider the Purdue //ELLPACK system, which is generally considered to be the state–of–the–art in this area.

The //ELLPACK PDE solving system [HRC+90, HR92] evolved from the ELLPACK [RB85] system. These systems have taken the approach of using a high level mathematical language to allow users to specify the particular mathematical model as well as how they want it solved. This high level specification is then translated to an imperative program (typically in FORTRAN) and then executed in conjunction with various solver libraries.

Taking this view, the ELLPACK–like systems decompose the PDE solution process to the following sequence of steps exhibited by many numerical PDE solution strategies: domain discretization, operator discretization, and (linear) equation solution. The PDE problem specification itself consists of the system of partial differential equations, the domain(s) of definition and possibly boundary and initial conditions. The high level languages in these systems therefore provide mechanisms for specifying the various PDE problem components and mechanisms for selecting a domain discretization scheme, an operator discretization scheme and a (linear) equation solution scheme. For more complicated problems (such as nonlinear or time dependent problems), these languages also allow one to specify various control mechanisms as well.

The //ELLPACK system evolved ELLPACK in many ways. Most importantly, //ELLPACK introduced a complete parallel environment for distributed memory MIMD machines consisting of parallel discretizers, solvers and other tools to allow users to apply these parallel components conveniently. In addition, //ELLPACK introduced a complete user–interface environment that afforded users significant assistance and convenience towards specifying the components of a problem and a solution process. This user interface environment consists of a collection of graphical editors, where each editor allows one to specify one component of the problem or the solution

process. The output of each editor is represented in textual form in the //ELLPACK language. When a complete problem specification and a solution algorithm specification is ready, the //ELLPACK program is given to the language processor for execution. Once the solution is complete, several other editors are available to the user, including visualization tools and performance analysis editors.

2.2.2 Application Development Frameworks

Application development frameworks for building PDE based applications are yet to appear. However, the idea of such frameworks for other application domains has been implemented in several cases.

ABE [ELHR88] is a programming environment supporting the abstraction "Module Oriented Programming" (MOP). MOP is somewhat similar to object–oriented programming, except that a module in ABE also has a thread of control. In MOP, an application consists of a number of asynchronous modules that can connect and communicate with each other in well defined ways. A module can be a "Black Box" module or a module consisting of other modules. Each module has a built–in controller that controls which sub–module runs at which time. Each controller may implement its only control philosophy (called "framework"), for example data flow, transaction processing or procedural. Sub–modules in a module communicate with each other via the network managed by the controller. A graphical interface called the "ABE Desktop" is available for developing, distributing, and executing applications. Each framework has its own graphical editor for viewing and editing modules composed of that type of framework. A catalog for storing and retrieving abstract data types and other ABE objects is available. A database interface provides connection to a relational database management system. ABE was designed to support the development of intelligent systems, where intelligent systems are defined as large–scale software systems that reuse and integrate knowledge–based and conventional systems. It has been used successfully in the development of a Pilot's Assistant program.

The ITHACA [AM90] object–oriented application development environment is a PSE framework designed for C* applications such as Computer Aided Design (CAD) and Computer Aided Manufacturing (CAM). The ITHACA environment is built atop the "HooDS Persistent Object Environment." HooDS consists of an object storage environment ("Noodle"), a debugger ("Max"), an application builder, a user interface development support system ("DialOOg") and a filter/browser tool ("FBi") for navigating the database. The basic idea for ITHACA is that a set of applications from a particular class share sufficient functionality to be built out of "generic" parts. These parts can be chosen from existing components (using the "Software Information Base" (SIB)) or implemented above the HooDS system. The hope is that the SIB is sufficiently rich to permit the parts implementation step ("Application Engineering", in their terms) to be smaller than the parts selection step ("Application Development"), resulting in more reliable software as well as greatly reduced development time. Several application specific workbenches consisting of SIBs have been

constructed.

Other application development frameworks include Explorer [Sil92], Agora [Bis88], Mathematica [Wol88] and Maple [Gro87].

2.2.3 Software Bus Environments

While the idea of a software bus is not recent, it appears that the implementation of the idea is recently becoming more and more popular (see for example [PAP+89], [SG93], [OPSS93], [HGJ+89] and [MV88]). We very briefly review the Polylith and Glish systems below.

The Polylith [Pur94] software bus is a software component decoupling agent. Software components are built using some abstract bus interface and the component's interface is specified using a "Module Interconnection Language" (MIL). The module interface definition is used to automatically generate stubs that can be used by other modules. Selecting a specific implementation of the abstract bus selects the mode of interaction between modules; whether it is message passing or shared memory, for example. Each component is not aware of the structure of the entire application itself; it has symbolic knowledge about itself and any other components it needs to interact with. The symbolic knowledge is transformed to actual locations at run–time by the software bus environment.

Glish [PS93] is a user–level software bus system implemented as an interpreted language for building distributed systems from modular, event–driven programs components. The Glish language is a full arrray–oriented programming language which can be used to manipulate events as they are being sent between clients. By default, clients connect with the Glish interpreter and a script specifies what is to be done with each event that arrives at the interpreter. The client programs are written in conventional languages and are linked with the Glish client library to get access to Glish. Clients communicate with each other by generating events. An event is a symbolic object consisting of a name and a value. Values can be arbitrary data structures and are communicated using a self–describing dataset format.

2.3 Design Objectives of PDELab

The advancements in high performance computing technologies have already allowed computational modeling to become a third form of scientific inquiry with the other two being theory and experimentation. It is expected that computational models will impact significantly both "big" and "small" science. In addition, in the near future we will see them used as training simulators of scientific and industrial processes and costly instruments. Their role in education will increase by using them to substitute many of the functions supported by the current wet/dry laboratories. Despite their positive impact and potential, their introduction has significantly increased the complexity of the scientific problem solving process. Figure 2.1 displays the PDELab view of the modern problem solving process for PDE based applications [Mac91b].

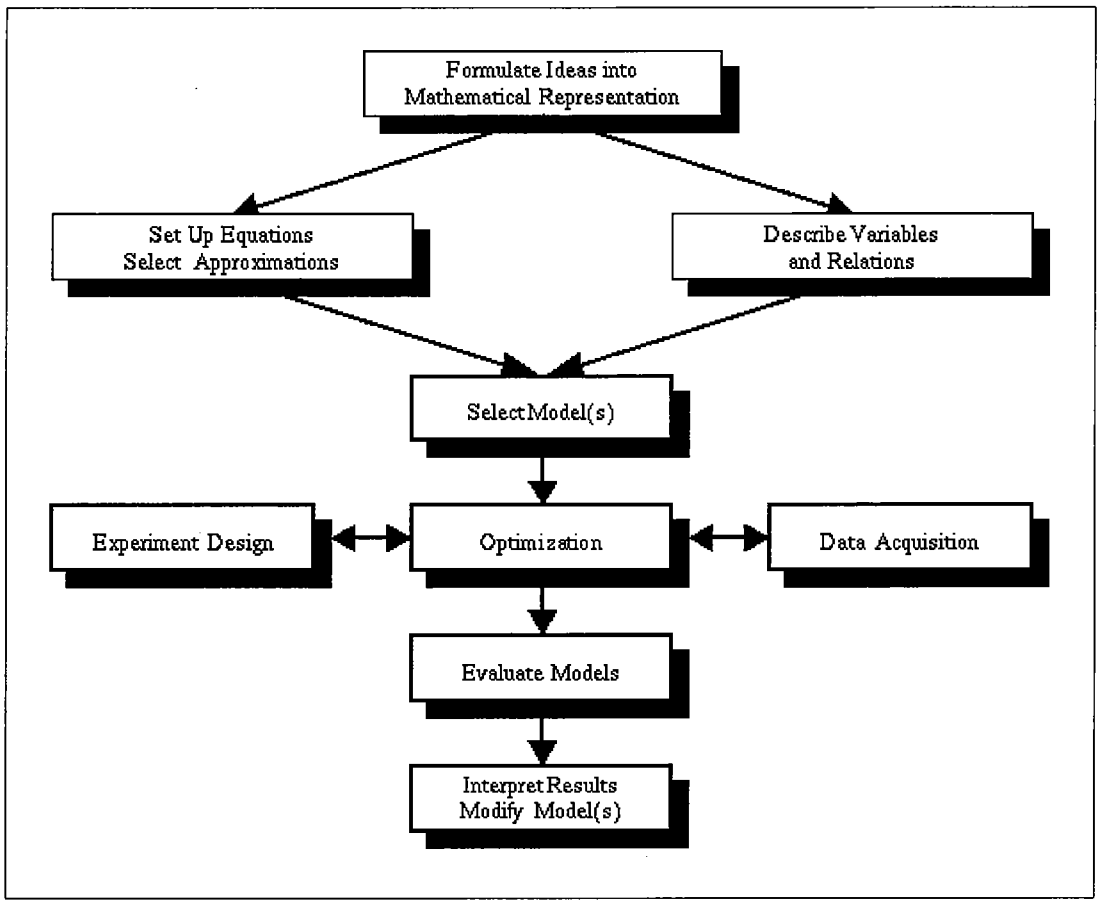

Figure 2.1: Scientific problem solving process for PDE based applications

It includes processes such as brain storming, trial and error reasoning, numeric and experimental data I/O and calibration, numeric and symbolic simulation, advanced reasoning, optimization, visualization and interpretation of results. The main design objective of PDELab is to emulate the above functionality of the problem solving process and automate many of its logical parts. For this we need to integrate numeric, symbolic, geometric, intelligent, and visual computing technologies. PDELab is expected to support users with different computational objectives, background and expertise. This implies that it must be intelligent enough to interpret high level queries whose processing will require support from scientific databases, knowledge bases, and information systems.

2.4 Software Architecture

The software architecture adopted for PDELab is characterized by the software independence of its parts. It is based on the "clean layering" and object-oriented methodologies. At the highest level PDELab consists of three layers. The lowest layer of this model represents the algorithmic and systems infrastructure needed to support

the numerical simulation of various "physical" objects on a variety of machines with virtual, generic and specialized architectures. In addition, this layer supports a high level memory system implemented through an object–oriented database system and various domain specific knowledge bases. Figure 2.2 illustrates this architecture. The remainder of this section describes the software architecture of each of the layers of PDELab and illustrates it with an application problem solving environment developed with the PDELab framework.

2.4.1 Algorithms and Systems Infrastructure

The lowest layer of the PDELab architecture consists of the libraries, knowledge bases and other similar computational agents that drive the simulation process. For PDE computing, these components manipulate a certain collection of meta–objects (consisting of code and knowledge) that are involved in PDE computations, including PDE equations, geometric domains, boundary and initial conditions, grids and meshes of the PDE geometric regions, matrix equations, and discrete functions. Each object consists of data structures for representing the data associated with the object, collections of functions that manipulate the data and associated knowledge that allows the object to perform various operations. The software architecture of this layer is therefore based on the structure and assumed interactions of these PDE objects. Each module of each library has a certain type signature based on these objects; that is, each module's input and output is given in terms of these objects. Internal to a module, the data structures and representations can be arbitrary, but that must be packaged into a standard form that uses the PDELab objects before a module can be integrated to PDELab.

The PDE objects are designed to be unifying representations of the PDE components; i.e., each object is expected to have sufficient flexibility to represent any instance of that type. This is achieved by following the mathematical behavior in the definitions and by studying the needs of various existing software packages. The unifying representation of an object is then defined as the union of identified requirements or as a classification based on the naturally occurring subtypes of the object. The latter form supports extendability. As software components are implemented in a multitude of languages, representations for the objects are defined in several languages. Currently, C, FORTRAN and Common LISP are supported. Functional representations (in addition to the data structure representation) are also defined when appropriate. As certain objects have multiple "standard" representations, alternate forms and conversions between them are also supported. The representation internal to a module is considered "private," but if the private representation is common, then a convertor to/from a public form can be registered for it. Convertors can be compiled–in functions or filters applied via pipes.

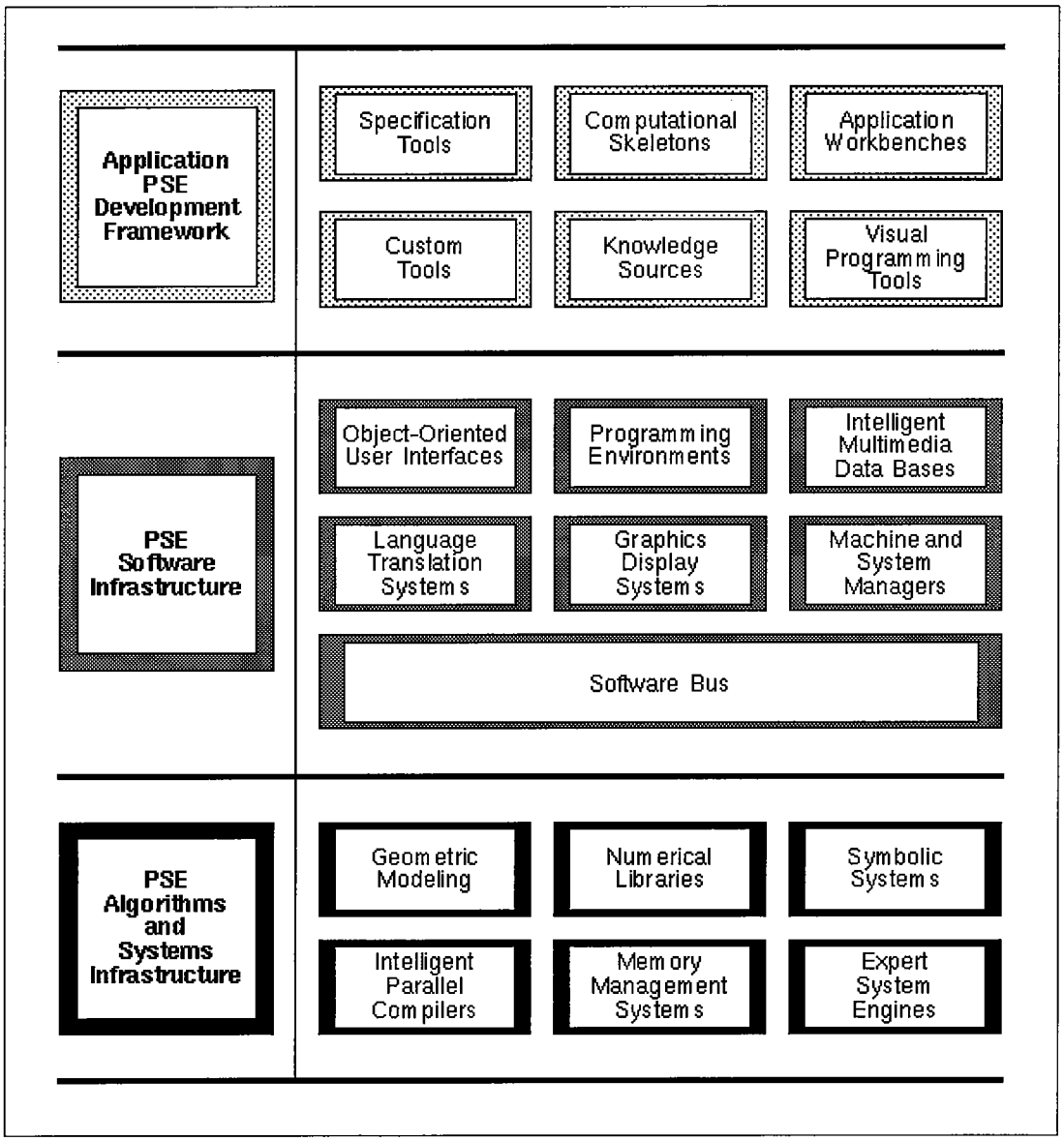

Figure 2.2: The layered software architecture of PDELab. The objects in the bottom layer are the PDE objects while objects in higher layers are programmatic objects possibly comprising of the PDE objects.

2.4.2 Software Infrastructure

The software infrastructure layer is the "middleware" that facilitates much of the functionality of PDELab. The task of this layer is to facilitate the integration of various software components to form integrated workbenches and also facilitate the development of these components as well. The communication fabric that facilitates component integration is based on the software bus model and is described in detail later in Section 2.5. Other software infrastructure needed to build the upper layer components include graphical display systems and user interfaces as well as object management and database functionality. The PDELab tool development kit, PDEKit, provides this functionality in the form of library utilities that component and custom application tool developers can use.

PDEKit defines and implements data structure representations for all the PDE objects in C, Common Lisp, XDR and FORTRAN. For those objects that have meaningful functional representations, they are defined for C, Common Lisp and FORTRAN. For the execution environment, run–time interfaces between various objects is also defined. Since these objects arise in many circumstances, they are allowed to have more than one representation in a given language. PDEKit supports this by providing representation convertors that allow one to easily convert from one representation to another. For example, PDEKit provides a convertor that translates a mesh given as a PATRAN Neutral File to the in–core mesh representation defined by PDELab.

The User Interface Development toolkit component of PDEKit is the implementation basis for the PDEView, the user interface environment of PDELab. This toolkit defines a standard look–and–feel for all the tools/editors and provides library level services that one can use to perform various user–interaction tasks. The toolkit adhere to the Motif User Interface guidelines and is built using the OSF/Motif Widget Set.

Other generic services provided by PDEKit include distributed file I/O mechanisms (using the software bus) and generic object management services to help manage collections of PDE objects.

PDEVpe is a framework within the PDELab environment which provides a platform for developing portable parallel PDE solvers, along with facilities and tools for analyzing their performance. PDEVpe includes parallel computational kernels such as the basic linear algebra subroutines (BLAS), portable communication libraries such as MPI [DHHW93] and PICL [GHPW92], parallel languages such as HPF [KLS+94], parallelizing methodologies for re–using sequential code in a parallel environment, geometry and matrix partitioning tools, instrumentation tools and skeletons (templates) for parallel methods such as PTS [WH94].

2.4.3 PSE Development Framework

This upper layer of the PDELab architecture provides application PSE developers with a collection of tools and services required to build such PSEs. The tools at this level include visual programming tools that support programming–in–the–large or

megaprogramming using PDELab components, computational skeletons for template based programming, tools for editing various PDE objects and support for building custom tools that deal with application dependent aspects of a PSE. These tools are organized into a toolbox available to the application PSE developer and appear as a collection of building tools integrated by the software bus. The tools themselves are built using the functionality provided by the lower layers and generate some form of configuration scripts. The software architecture of a tool is generally in the form of an event–driven process that is implicitly invoked by the software bus. Section 2.6 provides an overview of the tools provided by this layer.

2.4.4 Application Problem Solving Environments

The architecture of application PSEs developed using the PSE development framework views a PSE as a collection of distributed tools that collaborate with each other to solve some problem. This architecture is supported by the same software bus methodology and the collection of objects representing various PDE components that drives the PSE development framework. However, at the end-user level, the PSE developed using PDELab appears as a centralized system with the custom user-interface controlling all the components executing underneath. The custom interface communicates with the user in application domain terms (and not in mathematical PDE terms) by translating domain terminology to/from the appropriate mathematical representations.

2.5 Communication: The PDELab Software Bus

The underlying communication fabric for PDELab is based on the software bus [PSW91] model. The software bus concept is an attempt to emulate the hardware bus mechanism that provides a standard hardware interface to attach additional capabilities to a machine. In the hardware bus, new units describe their capabilities to the bus controller, which then passes the information along to other units in the bus. In the PDELab software bus, PDEBus, software components register their "exported" services with the software bus and rely on the software bus to invoke these services when requested by interested clients. The software bus is responsible for the application of any representation translators as required for the valid invocation of the service. Thus, the software bus provides a mechanism where two tools can interoperate with each other without having explicit knowledge about each other and also provides the infrastructure for managing a set of distributed tools.

2.5.1 Requirements: Clients, Protocols and Services

The requirements of the PDELab communication system can be stated in terms of three parameters: The types of clients it needs to service, the client communication protocols it supports, and the services provided to clients.

In the PDELab context, there are many types of clients (in terms of their execution nature) that must be supported. These include reactive or event–driven clients that register services that they provide and then asynchronously receive requests and service them. An example of this is a client that provides a some calculation service provider. Such a client would register the service and then forever wait for service requests and service them. Another type of client is a command–driven (shell–type) client; given some command, it responds with an "answer." Other clients are off–line clients which require no input, but some other client may be interested in its output. The software bus must provide tools and mechanisms to interact with all these types of clients as often we (as the developer of a problem solving environment) do not have the option of adapting them to any preferred interaction model.

PDEBus must support protocols for at least three different client interactions: the software bus' own client interaction protocol (for clients built with the software bus client library), raw byte-streams (for arbitrary communication) and a line-oriented protocol (for interacting with command-oriented clients). Other protocols supported may include the Shastra protocol, the communication protocol used in the Shastra project [AB94]. Supporting this protocol would allow PDELab clients to communicate with Shastra clients directly.

The services provided by PDEBus to clients can be categorized into three groups: location services, process management services and messaging services. For client and object location purposes, a global naming scheme based on uniform resource locators (URLs) [BL93], a highly flexible emerging standard for naming arbitrary resources, is used. The software bus provides various directory services with URLs being the naming standard. The process management facilities provided by PDEBus include facilities to invoke and control both local and remote processes and facilities to set up pre–wired configurations of clients and facilities. PDEBus' messaging services range from low–level byte stream messages to communicating arbitrary data structures via self–describing [1] and network–transparent representations to remote procedure calls. Using these communication facilities, application specific services (for example, a database service library) can be built for specific needs.

2.5.2 Bus Architecture

The PDEBus architecture reflects the problem solving environment architectural model described earlier. Clients are built using the PDEBus client library and, at run-time, connect to a "manager" process. A manager process exists for each user, application and machine (an "access domain") and serves as the clearinghouse of inter–client messages and client requests. While inter–client messages travel via the manager by default, it is possible to establish direct, point–to–point links when necessary. (Such direct links may be used when one wishes to communicate (large) data

[1] Self–describing representations allow receivers to interpret a raw byte sequences without having prior knowledge of the types of data being communicated. Using such representations makes it possible to build generic services (for example, a tracing facility for monitoring on–going communication) without being restricted to predefined data types.

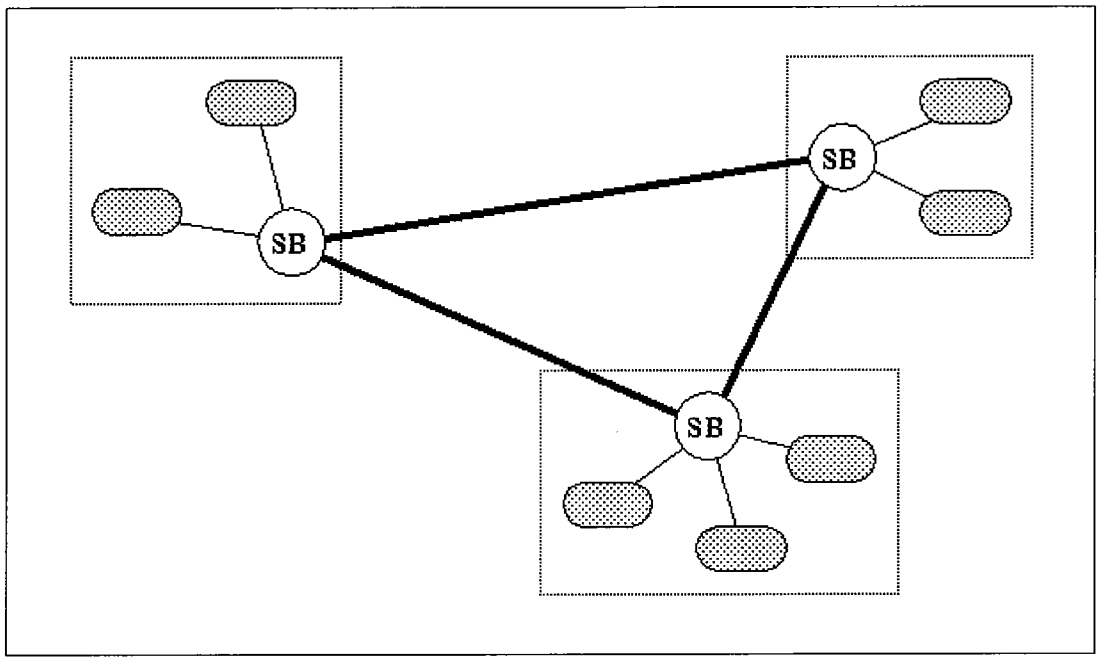

Figure 2.3: The architecture of PDEBus. "SB" represents a manager process, an oval shape represents a client while the dotted encapsulating boxes represent access domains.

without incurring the cost of travelling through one or more manager processes.) The manager processes themselves are connected to each other via multiple I/O channels. Interprocess communication occurs via TCP/IP sockets, pipes, shared memory or pseudo–terminals, depending on what the two components can support. In order to avoid being a bottleneck, the manager process generally uses shared memory to communicate with its clients and is multithreaded. Inter–client connections are virtual circuits connections implemented over the client–manager, manager–manager and manager–client connections. At any time, a client can have any number of virtual circuit connections to any number of clients (including itself). Figure 2.3 shows a schematic of this architecture.

The PDEBus implementation respects all the standard access controls supported by the underlying operating environment and guarantees security. This is affected by following the usual mechanisms for getting access to a machine (to run a manager process) and by using a key–based security mechanism for authenticating and validating clients once a manager process is active.

Communication of arbitrary data types is supported in two ways. First, a self–describing data format can be used to inform the underlying communication medium of the types of data being communicated. Second, PDEBus allows clients to register their own convertors to/from the data structure and some transport representation. Using this latter mechanism, one can transmit and receive data in the eXternal Data Representation (XDR [Man90]), for example. A set of utility functions for supporting XDR data communication is included in the current implementation of PDEBus.

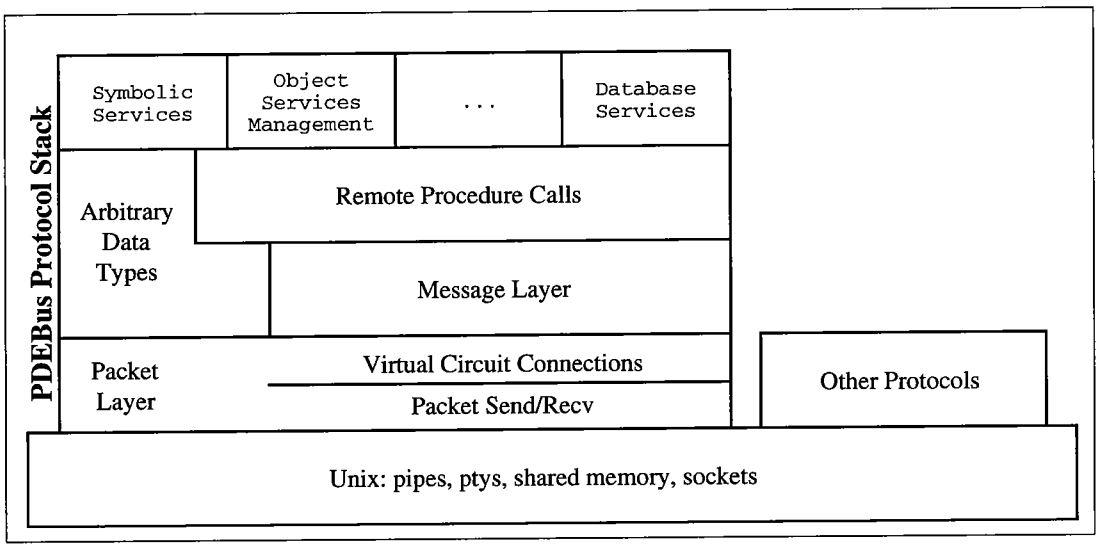

Figure 2.4: The software architecture of the PDEBus API library.

The software architecture of PDEBus is a layered architecture with the lowest level providing a packet–based messaging system implemented over a reliable byte–stream protocol such as TCP/IP. The next layer provides support for messaging and arbitrary data type communication. The highest layer provides remote procedure calls and event–driven messaging. Figure 2.4 illustrates the software architecture of the PDEBus application programming interface (API) library. This same library is used by PDEBus clients as well as the PDEBus manager.

2.5.3 Configuring a New PSE

Building an application problem solving environment requires one to develop application specific modules and then interconnect those modules with a subset of PDELab tools. Since inter–component communication is transparently achieved via URLs and PDEBus' messaging system, once the components are built, one only needs to have PDEBus initiate and manage those components. This is achieved via the session initiation script mechanism of PDEBus. This script (implemented using Tcl [Ous90]) instructs PDEBus to initiate the components required for the session. A graphical development tool at the upper layer of PDELab will assist in developing these session initialization scripts.

2.5.4 Comparison with Software Bus Reference Model

In [PSW91], Purtilo, Snodgrass and Wolff define a reference model for software bus systems consisting of three components: the abstract bus specification, the language used to interact with the abstract bus and the implementation of the abstract bus. In this section we apply this model to PDEBus.

The abstract bus specification consists of a type model defining the types of data

to be communicated on the bus and a control model that includes a common set of control paradigms by which control is transferred among the modules connected to the bus. PDEBus natively supports the standard basic types (integer, string etc.) and structured types (arrays and records) using a self–describing representation. Uninterpreted byte sequences are also supported and this is used to apply other linear data transport representations such as XDR. The control model of PDEBus is based on message passing. Both synchronous and event–driven asynchronous messaging is supported. A remote procedure call model implemented with the lower level messaging primitives is also provided.

The language clients use to interact with PDEBus consists of two components: a configuration language and an application programming interface. The configuration language allows one to specify the communication needs and behavior of clients as well as how the clients are to be (initially) interconnected. The transport representation of the communication data can be in any transport representation supported by the two communicating parties. Data that is intrinsic to PDEBus itself is communicated in network byte order for integral types and in XDR representation for structured types. PDEBus messages are reliable and have at–most–once delivery semantics with message order preserved between messages along the same virtual circuit connection.

The bus implementation consists of language bindings, the bus application programming interface, communication interrogation and the software bus environment. PDEBus' initial bindings are to C/C++. FORTRAN and Common LISP binding are also planned. The bus interface provides packet–oriented message passing at the lowest layer, a custom messaging protocol at the next layer and finally remote procedure calls implemented over this protocol at the highest level. For communication interrogation, clients have mechanisms to find out the status of communication links and perform other state queries and transformations. The software bus environment consists of a stand–alone process that manages the communication needs of a session, an application programming interface in the form of a library and several monitoring tools.

2.5.5 Why Yet Another Communication Library?

Given the recent proliferation of distributed object communication environments, the question "Why develop yet another system?" must be addressed. Let us classify existing distributed communication environments into three classes and then address this question in the context of each class:

- message passing libraries (e.g., PVM [BDG+92], PICL [GHPW92] and ISIS [Bir91])
- distributed computing environments (e.g., Sun's ONC [Sun91] and ToolTalk [Sun93] and OSF's DCE [Fou92])
- software bus systems (e.g., Glish [PS93], Polylith [Pur94])

The message passing libraries generally provide `send()` and `recv()` primitives for transmitting byte sequences between processes running on different machines. To transmit structured data types, it is necessary to build a layer above such libraries to first linearize such data (using XDR or some other mechanism) and then to transmit the resulting byte sequences. Also, for workstation clusters, these message passing environments tend to be explicit; i.e., the user must explicitly invoke and terminate the virtual multi–computer supported by these environments. Our application–oriented view requires an embeddable distributed communication environment that is transparent (or rather, implicit) to all parties involved in a communication as well as to the user.

Distributed computing environments provide much more functionality than do the message passing libraries. However, almost all expect that the components are built with knowledge of the communication facilities provided by them. In our case, we need to support legacy systems (for example, the MAXIMA[2] computer algebra system) without adapting them to a remote procedure call model, for example.

Software bus systems provide basically the same type of functionality that we have outlined for PDEBus. The only major missing feature is support for multiple client interaction protocols. While we probably could have extended one of these systems to meet our needs, we felt that developing our own environment is more suitable for our own long term goals with software bus type of environments.

A feature that appears to be common to every system is that they expect all the parties involved in a distributed session to be operating under the same protocols and conventions. While it is convenient to assume this when developing a distributed communication environment, it is certainly not necessary and is definitely a significant restriction when dealing with legacy systems.

2.6 PSE Development Framework

The highest layer of the PDELab environment is the problem solving environment development framework. At this level, application PSE developers compose new PSEs by combining together components from PDELab and the application specific components they implement using PDELab provided development tools. This framework consists of several subsystems: PDE object editing tools, a graphical worksheet editor, the PDESpec language and associated tools, the PYTHIA [HRWH94] reasoning environment, the composer, and of course the developer's kit described earlier. We discuss the object editing tools, the worksheet editor, PDESpec, PYTHIA and the composer here.

[2]MAXIMA is the Austin Kyoto Common Lisp version of the well–known computer algebra system MACSYMA[Gro77].

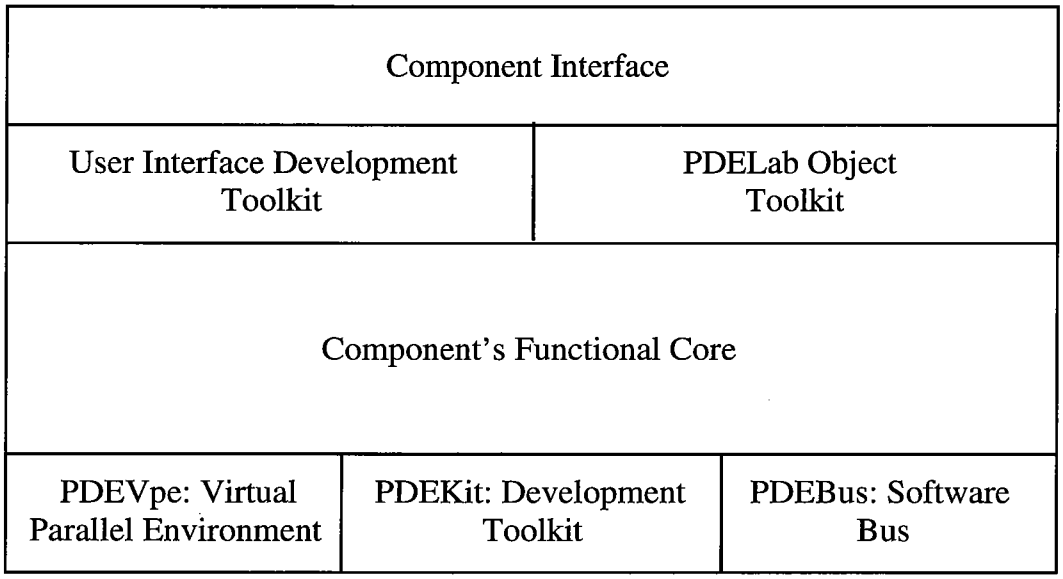

Figure 2.5: Software architecture of a PDELab editor. The functionality of the editor ("Component's Functional Core") could itself be built using facilities of PDEVpe, PDEKit and PDEBus. The functionality is incorporated to PDELab by building an interface using user interface development and object manipulation facilities of PDEKit.

2.6.1 PDE Object Editors

The PDEView object editing environment consists of a collection of tools that allow users to create, edit and manipulate PDE objects. For example, PDEView includes tools for editing domains, equations, boundary and initial conditions, generating meshes and visualizing data. The two–dimensional domain editor, for example, allows users to define a domain by specifying the boundary (graphically and/or textually) and any holes. The resulting domain object can be transmitted to another editor (to the mesh editor for meshing, for example) or can be saved in a file or in a session editor. Figure 2.5 illustrates the software architecture of a PDELab editor.

Several of PDEView's object editors are frameworks for integrating specific functionality relevant to the tasks supported by the editor. That is, an editor is not merely a user interface supporting certain built-in, fixed functionality. Instead, the editors support a process whereby libraries and systems that perform related operations can be installed in the editor. Thus, foreign systems and libraries can be integrated into PDELab with little effort and foreign software can take advantage of the complex PDE machinery built into the PDELab environment. As an example, let us consider the two–dimensional mesh editor.

The 2D mesh editor requires a 2D domain object as input and produces a mesh object as output. Given the vast array of 2D mesh generators that are available, it would be restrictive indeed to build in a fixed library of mesh generators. Instead, the mesh editor provides some basic functionality which includes loading in

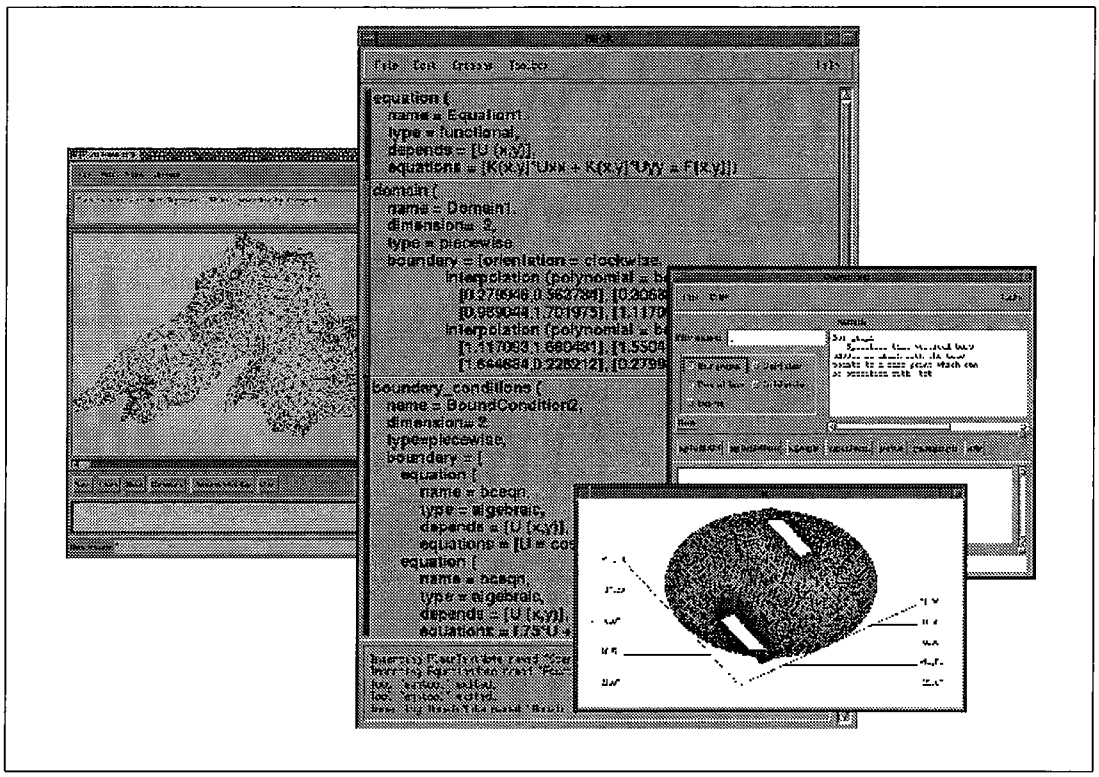

Figure 2.6: Some samples of PDEView editors.

and displaying 2D domains of various formats, displaying and saving 2D meshes in various formats and converting meshes from one format to another. The actual mesh generation capabilities are "loaded in" to the mesh editor via Tcl scripts. A mesh generator installer uses the embedded scripting language to instruct the mesh editor where the mesh library is, what top–level function to call, what the user–specifiable parameters are and what data the mesh generator produces. Format conversions can also be specified in this script. Adaptive mesh generators requiring multiple calls to the generators are also supported. Figure 2.6 shows instances of the PDELab 2D mesh editor, visualization controller, the 2D mesh visualizer and the worksheet editor described below.

2.6.2 Worksheet Editor

Application PSEs can use the graphical worksheet editor as the central access point for the PSE or as a session log containing all the objects created during the session. This editor provides standard editing capabilities (delete, cut, paste, etc.) and also allows users to convert between the text and data structure representations of objects.

2.6.3 PDESpec Language

The PDESpec language is PDE specification language that allows the user to specify the PDE problem using textbook type notation and to direct the PDE solution using a high level pseudo–code to combine existing computational modules or PDE parts in a serial or nested form. The language is also defined in terms of the PDE objects and the language form of an object is generally viewed as an alternate representation format for objects. PDESpec supports both a compiled execution model and an interpreted execution model and its syntax is defined as an extension of that of the MACSYMA [Gro77] language. The MACSYMA parser is used to parse a PDESpec program and the compiler (implemented in MACSYMA, Common LISP and C) generates a program in FORTRAN using the GENCRAY [WW92] code generation package. For interpreted programs, a parsed version of the program is handed over to the PDESpec interpreter for execution. By implementing the language as an extension of the MACSYMA computer algebra system's language, PDELab supports direct symbolic transformations at the language level. Chapter 3 discusses PDESpec in more detail.

2.6.4 PYTHIA Environment

In a problem solving environment, there are many situations where intelligent reasoning rather than algorithmic logic in necessary. For example, when selecting a solution scheme to solve a given PDE problem, it is necessary to apply various rules of thumb, heuristics and experience rather than a simple formula. PYTHIA supports this type of "soft" reasoning by providing an environment where one can integrate rule and knowledge bases and associated reasoning systems. PYTHIA is accessed as a tool in the PDELab environment and is implemented with an embedded version of CLIPS [Gia91]. Chapter 4 discusses PYTHIA in more detail.

2.6.5 Component Composition

A major step in building an application PSE with PDELab is combining a set of components together into a PSE using PDEBus. The composition editor supports this activity by providing a graphical environment for selecting components (editors) and "wiring" them together appropriately. The resulting data flow graph is transformed into PDEBus scripts that can be used to represent the composed application PSE.

2.7 Symbolic Computation in PDE Solving Environments

It has been widely recognized that integrating symbolic computation with numeric computation is beneficial to the process of solving partial differential equations (PDEs). In spite of this, most PDE solving software does not apply symbolic computation as

an integral part of the solution process. The primary reason for this lack of interaction between these two computing methodologies is the difficulty in achieving this interaction in practice. In this section, we highlight the convenient and ubiquitous integration of symbolic computation that we have achieved in PDELab. See [WHR94a] and [WCHR92] for details.

2.7.1 Symbolic Computation in PDELab

As mentioned above, the PDESpec language is defined using MACSYMA. The PDESpec parser that is invoked when users textually enter PDE objects is implemented within MACSYMA. Symbolic transformations on PDESpec objects (for example, linearizing a nonlinear equation or discretizing a PDE operator) are also supported. When a PDESpec program is to be executed, a translator in MACSYMA symbolically analyzes the program and generates the target code using the GENCRAY code generation system [WW92]. Execution time discretizations (for example, discretizing continuous boundary conditions to generate discrete values at boundary points) are done using MACSYMA as an evaluation server.

Within PDEPack, direct symbolic operations are not feasible due to the difficulty of efficiently integrating the run–time mechanisms of languages such as C and FORTRAN with that of a symbolic computing environment. Instead, PDELab allows applications to statically specify the symbolic operation to be performed, but have it invoked dynamically. The static specification allows us to allocate necessary storage and control mechanisms while the dynamic invocation provides all the necessary flexibility to applications. Jacobian generation is an example of this type of usage.

Several PDEView tools apply MACSYMA as a dynamic evaluation mechanism. For example, the domain editor uses it to generate a display representation of a boundary defined in a piecewise parametric form: the parametric expressions are evaluated at various parameter values to produce a piecewise linear representation that can be displayed. The domain editor also uses this mechanism to allow users to specify arbitrary interpolation schemes. The same mechanism is used to generate piecewise linear representations of the boundary needed by certain mesh generators. In the visualization environment interpolation, differentiation, smoothing and function overlays are supported by using MACSYMA.

PYTHIA, the computational intelligence environment uses MACSYMA to determine various properties of the equations, boundary conditions and initial conditions in order to provide advice on solution methods. Properties derived using MACSYMA include coefficient properties, dimensionality, boundary condition types and also more subtle properties such as singularities within the coefficient space.

The above uses of symbolic computation are all within the PDE solution environment itself. However, symbolic computation also forms an integral part of the overall problem solving process that is supported by the problem solving environments developed using PDELab. The PDELab system also provides a general environment where one can perform symbolic manipulation independent of the PDE processes involved

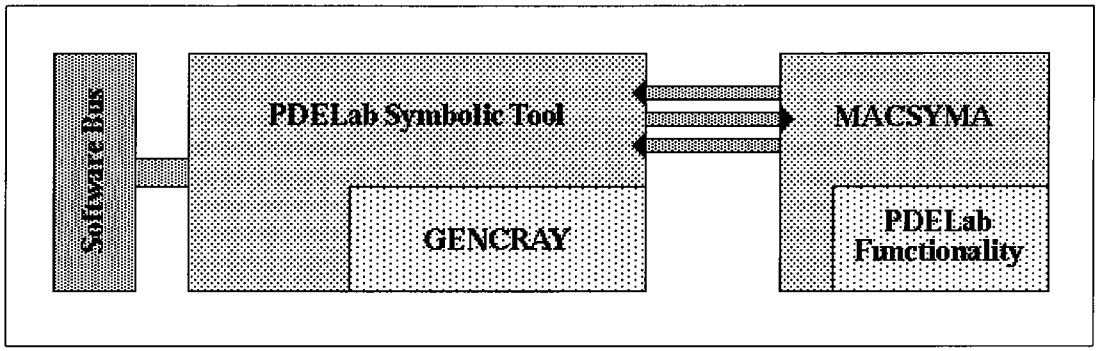

Figure 2.7: The architecture of the PDELab symbolic tool.

and incorporate these results into the worksheet editor as well.

2.7.2 Implementation

Since PDESpec is defined using a symbolic model, most symbolic transformations on PDE operators and boundary conditions, etc., are directly representable in PDESpec. In some cases, transformations can produce not only primitive objects, but entire program segments (in the form of algorithm objects) as well. The interaction in terms of the symbolic objects of PDELab is the key to providing arbitrary symbolic computation facilities to the PDE solving environment.

To provide symbolic computation facilities to all interested tools, a special tool is employed as the symbolic computation server. This tool exports a set of services accessed by interested clients. As we use the MACSYMA computer algebra system to implement the symbolic manipulation, the symbolic tool is implemented by front-ending MACSYMA. In addition, this tool provides program generation services using GENCRAY. Figure 2.7 illustrates the architecture of the symbolic tool of PDELab.

When a symbolic computation service is requested via the software bus, this request is transformed into one or more MACSYMA calls by the symbolic tool. The MACSYMA calls may be native calls or calls provided by the functionality specifically implemented for PDELab. Arguments to these calls may be arbitrary PDE objects and are transmitted to MACSYMA via a communication link in a textual form. The normal output produced by MACSYMA (for example, status and warning messages) are captured by the symbolic tool and displayed to the user. The communication protocol used in these input/output streams is the text command language used by MACSYMA. The results of the request, however, are sent to the symbolic tool via a special stream using the normal software bus communication protocol. The symbolic tool may further process these requests (for example, the message may be a program in an intermediate form that needs further processing by GENCRAY) or directly forward the reply to the original requestor. Program translation services of GENCRAY are provided by directly embedding GENCRAY in the symbolic tool.

To illustrate this scenario, let us consider an example. Suppose that PYTHIA wants to determine the properties of a PDE operator. PYTHIA sends the request

"operator properties" to the symbolic tool with the operator object as an argument. The symbolic tool translates the operator to a text form and calls the PDELab specific function that provides this service. This function determines the properties it can determine via various symbolic transformations and sends a reply to the symbolic tool. In this case no further processing by the symbolic tool is necessary; hence it simply forwards the list of properties (in the form of an object that the requestor can understand) to the original service requestor.

2.8 Example: Integrating a New Solver

To illustrate PDELab's flexibility in terms of integration capabilities, we describe an example of integrating a new PDE solver into PDELab. The solver under consideration is a FORTRAN implemented Navier–Stokes equations solver capable of handling incompressible fluid flow problems. The type of Navier–Stokes equations handled by this particular solver is a system of two, coupled, 2nd order equations in two space dimensions. This finite element solver expects an 8–node quadrilateral mesh and allows a small number of parameters in the equations themselves and supports either Dirichlet or Neumann boundary conditions for each of the two unknowns. The boundary conditions must be specified discretely at each boundary point and the solver also needs to be told what type of boundary condition holds at each point.

A custom template[3] was built to allow user's to enter the parameters of the equations handled by this solver. The output of this template editor is an equation object containing the Navier–Stokes equations with special properties indicating the values of the interesting parameters. The problem domain can be graphically or textually specified using the existing PDELab 2D domain editor. Boundary conditions are also specified at the continuous domain level (i.e., at each of the boundary pieces) and represented in the continuous boundary condition object format. In order to generate the 8–node meshes for this solver, we modified an existing quadrilateral mesh generator. (These modifications augmented the functionality of this mesh generator in a general way.) This mesh generator required the domain specified in a piecewise–linear fashion; hence the piecewise–parametric representation used by the 2D domain editor is automatically converted to the linear form before invoking the mesh generator. Once the equation, domain, boundary conditions and mesh have been defined, one selects the template algorithm object also generated by the Navier–Stokes equation template and invokes the *composition tool* to create an instance of the solve object that specifies the algorithm to be invoked as well as the parameter objects. The composition tool is used to compose a problem to be solved; it allows the user to compose an algorithm object together with the necessary argument objects of that algorithm. By examining the property list associated with the solve object, the *execute tool* determines that the Navier–Stokes driver should be invoked. The execute tool provides execution and monitoring services to the user.

[3]A template is an editor that produces one or more PDE objects. However, the user can only control certain parameters of those objects, hence the use of the term template instead of editor.

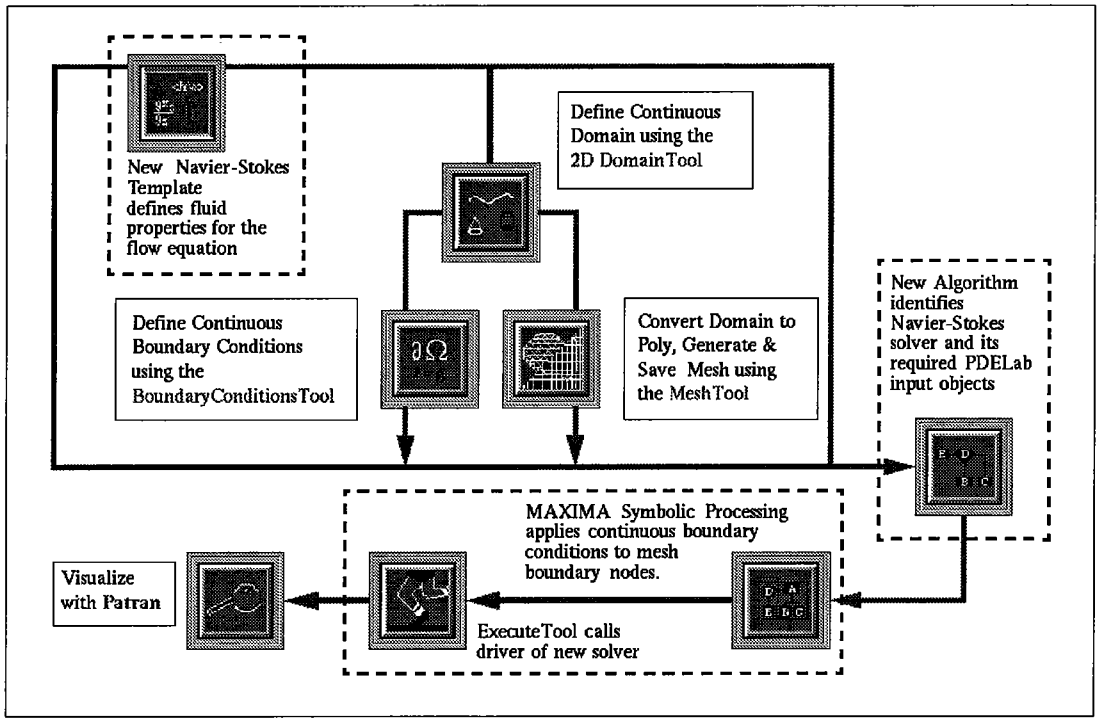

Figure 2.8: The Navier–Stokes solver integration process. The dashed boxes represent new or augmented components. Other unmodified components used are indicated by the unboxed icons.

Once the Navier–Stokes driver is invoked, it determines that the continuous boundary conditions must be evaluated at the boundary points of the mesh to produce a discrete representation. It also needs the type of boundary condition that holds at each node. Service requests are then sent to MAXIMA to evaluate the boundary conditions and to return the value(s) and type(s) at each boundary node. To support this functionality, custom code was added to MAXIMA and is invoked using the standard service invocation method. Once all the necessary data is available, the appropriate environment is set up and the FORTRAN solver is invoked.

After the solution is computed and saved, a custom translator is called to load the solution data into a solution data object for postprocessing using the PDELab visualization environment's tools.

Figure 2.8 illustrates this integration process. By integrating the Navier–Stokes solver to PDELab we have demonstrated the viability of the PDELab model of building problem solving environments. This example clearly shows how existing tools of the PDELab environment can be coupled together with custom code to produce an application specific environment.

2.9 Example: The Bioseparation Workbench

A major objective of PDELab is to provide a methodology for building application–specific problem solving environments. This methodology allows users with very specialized requirements to wire together components of PDELab to create an environment exactly suited to the solution of the PDEs that model their scientific application. The Bioseparation Workbench, referred to throughout as BioSoftLab, is a prototype of such a specialized environment which is currently being developed using PDELab. Bioseparation is the process for separating a compound solution into its constituent parts by passing it through an absorbent column so that each component adsorbs to the surface differently from the others, and thus eludes at different times [BWZW91]. This process is used in the purification of proteins and biochemicals, in the manufacture of pharmaceutical products, in water treatment, and in many other biochemical processes. The physical behavior of this process is mathematically modeled by a system of 1–D and 2–D nonlinear, time–dependent PDEs. The goal of the BioSoftLab PSE is to model the bioseparation process. Users of BioSoftLab are (practicing and apprentice) Chemical Engineers who perform bioseparation experiments. The results of these simulations are typically used to augment the data achieved via experimentation and also to predict the behavior of the experimental elusion process and thereby providing more insight into this highly nonlinear process. code.

The BioSoftLab provides a powerful and flexible environment for simulating the experiment numerically, using all the editors, techniques and solvers from PDELab that are appropriate to the model for this process. This workbench supports the specification, solution, and analysis of the bioseparation model using many different schemes, including the original customized code.

The PDELab methodology was first used to integrate the original solution code into BioSoftLab, so that the engineers could continue to simulate the process as they have done in the past. PDEKit development tools were used to build a template for entering numerical input data specific to their customized solver. The template is an abstraction of the PDE model, and communicates in terminology that is familiar to the engineer, an important goal of the PDELab model. The original solution software was then integrated into the PDEPack library of solvers, and is accessed through the BioSoftLab template. In this example, none of the problem specification object editors of PDELab are used; the template itself generates the complete problem specification for the solver. The composition editor and the execute editor are still used however. Once the simulation executes, results are generated in various formats. Custom translators were implemented to load in the results into solution data objects. Then, PDELab visualization tools were used for viewing time slices, data slices or animations of the results. This procedure for solving the bioseparation problem used the PDELab template builder, PDEPack solver integration techniques, several PDELab editors and the PDELab visualization tools. The engineers have gained a graphical interface for problem specification and solution, and visualization

tools for viewing and analyzing the simulation results.

The solution of the bioseparation PDE models for a realistic simulation often requires more than 20 hours of execution time on Sun SparcStation 10 processors. Since PDELab provides numerous tools and methods for defining and solving PDE problems, it is natural to investigate other solution schemes for the bioseparation process with the hope of using either faster sequential methods or parallel methods. The second strategy can be achieved by replacing the ODE solver used by the original code with a parallel ODE solver. The parallel solver could use the parallel execution services of PDELab and be called as a foreign solver from the bioseparation solver.

The bioseparation model can also be solved by the combination of two solvers available in PDELab. Since one is a 1–D solver and the other is a 2–D solver, merging these two into one application is not a trivial process. In this case the PDE operators, domains and boundary conditions are all specified using the standard PDELab objects. Then, a custom algorithm object that iteratively applies the two solvers until convergence is used to drive the solution process. The difficult task is how to translate this object into the PDELab execution environment. This issue is addressed in Chapter 3.

2.10 Conclusion

In this chapter we describe an application development framework for building problem solving environments for PDE based applications. The objectives, software architecture and implementation of the PDELab are described. The base architecture is based on a set of PDE–specific meta–objects that are intrinsic to the PDE solution process. Components of the framework interoperate by communicating these objects over a transparent communication medium called a software bus. Two example usages of PDELab are given to illustrate the functionality and applicability of PDELab. As shown by the ease with which the two example application PSEs were built, we see that the PDELab environment satisfies many of the design goals described earlier.

Chapter 3

High Level Language for Partial Differential Equation Based Mathematical Models and Their Solutions

In this chapter we present a high level symbolic language for specifying partial differential equation problems and their solution schemes. The language is based on decomposing the PDE problem into its constituent parts and on decomposing the solution process into a sequence of transformations. By integrating symbolic and numeric computing, we develop an environment that presents powerful facilities for composing new solution schemes from existing software components. The rationale for the approach we follow as well as the design and implementation of the language is described. Several examples are provided to validate its generality and flexibility.

3.1 Introduction

While interactive, graphical tools are very useful in specifying PDE problems, it is also necessary to have a concise mathematical representation in a form that is easily understood. Having a textbook–like notation for writing PDE problems and their solution schemes often allows users who are familiar with this notation to describe problems easily without having to resort to complex graphical tools. Furthermore, textual representations allow easy modifications to produce similar models. In terms of solution schemes, having a language for expressing algorithms allow users to combine PDE solving software parts to build complex solution strategies that were not previously available [HRW94, Ric89].

It has also been widely recognized that symbolic manipulation is an essential component of the process of solving PDE problems.[1] A high level PDE specification

[1]See for example the many special sessions organized on symbolic computing for PDEs in the

language must be inherently symbolic in terms of how it deals with the PDE problem. That is, the language should allow users to manipulate the problem components symbolically in order to allow for greatest flexibility. Symbolic manipulation of PDE operators, boundary conditions and even domains can transform seemingly intractable problems into problems that can be solved for example, by some iterative composition of existing solvers.

High level languages for PDE problem and solution scheme specification are almost as old as are high–level languages themselves [CK70]. More modern languages include the ELLPACK family of languages [RB85, HRC+90, BD90], the closely related PDE/PROTRAN language [IMS86], the DEQSOL language of PDEQSOL [Hit90] and others such as the object–oriented model developed by Ruppelt and Wirtz [RW89] and more application dependent systems such as Thornburg's PDE compiler [Tho89]. Most of these languages are non–symbolic in their manipulation of the PDE problem and tend to be direct translators from the mathematical representation to a run–time representation in a traditional imperative language. There are also systems that take a purely symbolic view; see for example [JP92] and [Wan86]. In these systems, the mathematical models are translated into an imperative language by performing most of the discretizations directly in the symbolic environments. That is, they tend to apply a pure program generation approach rather than the (mostly) pure program translation approach taken by the other systems. A feature that is common to all of these systems is that the types of models that they deal with is restricted to whatever their numerical solvers were capable of handling.

Our final goal is the development of a textbook–style natural language for specifying PDE problems and their solution strategies. As an intermediary to this goal, in this chapter we develop a formal language with sufficient expressiveness to encompass a wide range of problems. This language can be viewed as an "assembly language" for PDE compuatations and (human) users are expected to work with higher level tools and natural languages.

As a representation language, whether the problems can be solved by available solvers or not is not an issue; we are simply concerned about representing arbitrary problems and their solution schemes in a concise, precise form. To achieve this goal, we use symbolic representations of the components developed with full generality in mind. The second goal is, of course, to be able to execute a PDESpec program to solve the problem at hand. At this stage, we use a mix of symbolic and non–symbolic transforms and representations to generate an executable code that can be used to solve the problem. We, of course, do not have solvers for all problems; we handle the cases that are currently tractable and leave hooks for the others.

The ultimate goal is a symbolic specification environment for handling general integro–differential equations. We restrict ourselves to differential equations in this version of PDESpec. Also, as complex PDE problems generally have multiple regions of interest, the language must have support for multi–region equations, domains, etc.. Again we restrict ourselves to single–region problems for the most part in this work

recent IMACS World Congress' on Computation and Applied Mathematics.

and view multi–region problems as a composition of single–region problems (which is still possible in the current framework). For the types of numerical solution schemes considered, we restrict ourselves in this version to finite difference and finite element type solvers.

There are two components to this work. First, we address the problem of a high level language for representing partial differential equation based mathematical models. Second, we address the problem of how one specifies in a high level manner the scheme with which a given set of models are to be solved. This chapter is organized as follows: Section 3.2 looks at some related work in more detail. Section 3.3 discusses the approach we are taking and the rationale for it. Sections 3.4, 3.5 and 3.6 consider the representation of various components of the PDE problem and solution scheme. Section 3.7 discusses representation of PDE solutions. Section 3.8 discusses the syntax and semantics of PDESpec while the following section discusses the run–time system and other implementation issues. Section 3.9 discusses implementation issues. Section 3.10 highlights the symbolic manipulation done in PDESpec. Finally, Section 3.11 provide some examples that illustrate some of the salient features of PDESpec.

3.2 Related Work

In this section we briefly review some of the languages that have evolved over the last 25 years and highlight the features of interest. We consider first those languages that have taken a program translation approach and then those that have taken a program generation approach.

3.2.1 Program Translation–Type Languages

The earliest high–level PDE specification language is PDEL, the Partial Differential Equation Language [CK70]. The goal of PDEL was to significantly ease the (PDE) problem solving process by allowing users to specify the problem in a high level syntax which is automatically translated to a program in some other language. PDEL was implemented (in PL/1) as a preprocessor that translated a PDEL program to a PL/1 program that invoked library functions to solve the problem at hand. The language itself was developed as an extension of PL/1, allowing experienced PDEL programmers to write complex programs that included direct PL/1 code. The language supported a wide range of PDE operators over 1, 2 and 3 dimensional rectangular regions and generated tabular output and even contour plots of the solution.

The ELLPACK project [RB85] is a high level problem solving environment for solving a single elliptic PDE. The ELLPACK language is a high level language similar to PDEL. It is viewed as an extension of FORTRAN and, similar to PDEL, allows experienced users to insert arbitrary FORTRAN code to create complex programs. The ELLPACK model views the PDE problem in terms of its intrinsic components (equation, domain and boundary conditions). In addition, it decomposes the PDE

solution process into the distinct steps of domain discretization, operator discretization, linear system reordering, linear system solution and output. Users are allowed to select appropriate modules for each such phase, leading to literally hundreds of possible solution paths for a given problem. Standard interfaces are defined between these modules with the goal of being a repository for contributor provided modules. The language is implemented as a translator that generates a FORTRAN program which is them linked with pre–written libraries to solve the problem at hand.

The //ELLPACK[2] project [HRC+90] started with extending the ELLPACK system to support parallelism but has evolved into a complete interactive problem solving environment. The //ELLPACK language is an extension of the ELLPACK language and includes extensions for dealing with domain–decomposition based parallel solvers and for finite element methods (ELLPACK's data structures and language were designed for finite difference methods and structured finite element grids only). The basic execution philosophy of the language was not changed however.

The PDE/PROTRAN [IMS86] system is a high level problem solving environment for solving systems of linear or nonlinear elliptic and parabolic PDEs in general 2–dimensional domains. The PDE/PROTRAN language is similar in spirit to the ELLPACK language and is implemented as an extension of FORTRAN. The language includes constructs for defining arbitrary 2–dimensional domains, for defining the PDE operator equations and for mesh specification. The solution scheme is finite element type and all iterations, etc., are internally performed (i.e., there is no user level iteration scheme specification). The language translator generates a FORTRAN driver program which is linked with libraries to solve the problem.

3.2.2 Code Generation Based Languages

PDEQSOL [Hit90] is a Japanese project with a similar vision as the ELLPACK and //ELLPACK systems. The DEQSOL language is a complete language that is compiled into vectorized FORTRAN by a compiler that performs various symbolic transformations on the operators. The given PDE operator is symbolically discretized and the FORTRAN code to generate the discretization matrix is generated by the compiler. (The compiler typically generates many thousands of lines of code for a given problem.) Both finite difference discretizations and finite element discretizations are supported. The language syntax allows PDEs to be expressed in a natural mathematical syntax. While only linear elliptic operators are automatically discretized, problems with other types of operators may be solved via user implemented iterations. Two–dimensional domains can be described textually in PDEQSOL while an interface to a CAD system is supported for 3–dimensional domains. The compiler is implemented in Pascal.

ALPAL (A Livermore Physics Application Language) [JP92] is a system that automatically generates FORTRAN or C code to solve nonlinear integro–differential equations. The ALPAL language allows users to specify the equations in a symbolic

[2] "//ELLPACK" is read as "parallel ELLPACK."

manner and applies difference operators and quadrature rules to generate the resulting code. Logically rectangular grids on arbitrary dimensions are supported with either built–in or user–defined difference operators. ALPAL also generates Jacobian matrices and allows the use of external packages for solving systems of equations, for example. The system is implemented in MACSYMA.

SINAPSE [KDMW92] is a program synthesis system that generates numerical code for solving PDE or matrix equations using symbolic discretization. SINAPSE is implemented on top of Mathematica [Wol88] and uses a custom syntax for describing the PDE operators. The mathematical descriptions are then translated into an abstract algorithm description targeted to a specific machine architecture. The abstract algorithm (expressed in terms of difference expressions and regions on a grid) is then translated to a generic program by expanding differences etc.. Finally, the detailed representation is converted to C or FORTRAN.

3.3 Approach and Rationale

The basic approach we have taken in PDESpec is based on decomposing the PDE problem to its constituent parts of equation, domain, boundary conditions and initial conditions. Then, we view the process of solving PDE problems as the application of a sequence of transformations to the problem where each step reduces the problem to an "easier" problem, until the desired result is obtained. Hence, our strategy is to first develop a unifying notation for specifying partial differential equation problems. Then, we develop a notation for describing, at a very high level, the desired solution strategy as a sequence of transformations. Finally, we develop techniques for executing these transformations in order to produce the desired results. The rationale for this approach is that this represents both the mathematical methodology for solving PDE problems as well as the methodology used by existing software. (The methodology often gets lost in the details of imperative language constructs in a typical software package, but the abstract representation of the process does not change.)

Thus, in order to achieve our goals, we need representations of all the problem components as well as an understanding of the various transformations that are possible and what each transformation does. Since mathematics is a symbolic domain, it is natural to take a symbolic view at the highest level and develop symbolic representations for the problem components. To develop such representations, we study the mathematical formulation of PDEs as well as the needs of existing PDE software and attempt to produce a unifying representation. Similarly, to understand what the valid and useful transformations are, we study the related mathematics and PDE software to develop a unifying collection of symbolic and non–symbolic transformations. Then, once all the valid combinations of transformations are understood, we develop a consistent compile–time and run–time environment that will allow us to apply these transformations and produce the desired results.

As exemplified by ALPAL and SINAPSE (above), it is possible to generate the entire PDE solver code by symbolically representing all the transformations involved.

However, we use a mixed approach here; PDESpec's transformations are performed only up to a certain amount of detail. Below that point, we assume that black–box software has been produced and is available for use. We are not asserting that a total code generation style approach is not appropriate; we are simply choosing an arbitrary cut–off point for that approach. The advantage to our style is that we can reuse the large amount of existing software, while the total generation approach has the potential for being more efficient with the ability to generate highly optimized, custom code. A future marriage of these strategies will undoubtedly provide the most flexibility to the user.

We have so far only concerned ourselves with specifying problems and solution schemes. The important missing component is the representation of PDE solutions. The issue of how a PDE solution is represented becomes vital in complex solution methods based on iteration schemes such as interface relaxation and when one attempts to use software components from one package in conjunction with components from another package with different (internal) representations. Hence we also develop a general methodology for representing PDE solutions so that they can be reused as needed.

3.4 Specification of PDE Models

A partial differential equation model consists of four components: the PDE operator equations, the domains of definition of the equations, initial conditions and boundary conditions. The PDE operators hold over the interior of the domain(s) and boundary conditions hold over the boundary of the domain(s) at all time after some given "initial" time. The condition of the system at the initial time over the interior and boundary of the domain(s) is given by the initial conditions.

In this section we consider the structure of these "Meta–Objects" and attempt to identify their attributes. Our goal is to understand the mathematical structure of these objects[3] in order to develop representations that are sufficiently flexible to represent a wide range of PDE models. Later we validate the structure we identify by using our objects to represent models from various PDE solvers and applications.

3.4.1 Domains

Symbolic specification of geometric regions (domains) is an extremely difficult problem to solve in general in a dimension–independent manner. However, since our interest is primarily in 1, 2 and 3 dimensional domains, we can develop dimension specific, symbolic and semi–symbolic representations that satisfy our needs in each of these cases. Below we describe both the structure and the language form for each case.

[3]We use the term "object" here in the abstract data type sense rather than in the object–oriented sense.

The representation of 1–dimensional domains is straightforward: a single region is represented by an interval and multiple regions by a set of intervals. Regions can be coupled at the boundary of two intervals or can overlap. We use a pair of values to represent 1–dimensional regions. A 1–D domain object has as attributes the dimension of the domain (1) and the end points of the domain.

Two–dimensional domains are more challenging and there are several alternatives for viable symbolic representations. Explicit formulae (i.e., parametric representations, e.g., $x = \cos(t), y = \sin(t), t = 0 \ldots 2\pi$) are common and very flexible. Implicit formulae (e.g., $x^2 + y^2 \leq 1$) are generally more concise than their counterparts, but typically are expensive to use later for mesh generation and so on. Other common approaches are various boundary representations, both explicit (e.g., polygonal schemes) and implicit (e.g., $x^2 + y^2 = 1$). A 2–D domain object has as attributes the dimension of the domain (2), the type of representation being used (e.g., parametric, implicit and polygonal) and the representation specific information.

The representation of 3–dimensional domains is considerably more challenging than the 1– and 2–dimensional cases. While there are standard representations (for example, CSG or Constructive Solid Geometry) for representing solids, there are limitations with regards to using them later for mesh generation, for example. A less space efficient representation is a surface polygonalization. Surface parameterization is another useful representation. We allow for an open ended set of types of representations. A 3–D domain object has as attributes the dimension of the domain (3), the type of representation being used (e.g., CSG, surface polygonalization or surface parameterization) and any representation specific information.

Domains are defined in some coordinate system, which is noted for later use during plotting, for example. Domains also have a "unit" attribute (e.g., inch or centimeter) that associates physical measures with the numbers being used. If all components of the problem are specified using the same units, then essentially the system is dimensionless and the unit attributes can be ignored. However, if different units are used, then care must be taken to transform all interacting components to some common unit before operating on them.

3.4.2 PDE Operator Equations

Partial differential equations can be represented in several forms. The general (partial or ordinary) differential form is as a nonlinear function of the unknowns, the derivatives of the unknowns and the independent variables. In the variational formulation, the coefficient functions of all the components of the variational formulation must be entered as functions of the independent variables (see [GS91] for details). Other forms (which PDESpec does not as yet support) are integro–differential equations, and linear and nonlinear algebraic equations.

Once the equations themselves have been specified, one must define the domains over which they hold. For each independent variable of the equation, the domain of definition must be specified by indicating the name of an appropriate domain ob-

ject. The simulation time for transient problems is represented as a 1–dimensional domain. Other attributes of equations are dimension (1, 2 or 3), spatial and temporal independent variables, solution component dependencies (i.e., which independent variable(s) each solution component (or dependent variable) depends on), coordinate system of the (spatial) independent variables, type (differential, integro–differential or algebraic) and aliases. Aliases are a convenience mechanism for users to define commonly occurring operator forms and other macros, for example.

3.4.3 Boundary Conditions

Boundary conditions are (differential) equations that hold over the boundary of the spatial domain of definition of an equation. Different boundary conditions are often applied on different pieces or patches of the boundary and each condition consists of as many equations as there are in the system of equations that holds in the interior of the domain. Hence boundary conditions are dependent on both a domain as well as an (interior) equation.

3.4.4 Initial Conditions

In the case of transient problems, initial conditions describe the state of the system at the initial "time." They are expressions that specify values for the unknowns of the associated equation object at the initial time over the interior and boundary of the domain.

Equations, boundary conditions and initial conditions are generally specified on the continuous interior domain, patches of the continuous boundary and the temporal domain, respectively. However, sometimes it is necessary to specify them on discrete representations of the domain such as meshes. In this case, the discrete points can be grouped into collections where each collection is considered a continuous patch of the domain/boundary/temporal domain. Then, the equation, boundary condition and initial condition specifications are exactly as in the continuous case.

3.5 Discrete Geometries and Geometry Decompositions

While continuous geometry descriptions are useful as a specification mechanism, most numerical solvers work on discrete representations of the domain of definition of the problem. In this section, we consider the some issues in representing discrete geometries for use by PDE solvers.

A mesh of a domain is a partitioning of the domain into many small "elements" or regions such that the union of these elements covers the entire domain while satisfying certain geometric properties. Our interest in meshes is to identify a characterization

of various types of meshes and the uses of meshes in PDELab in order to develop an efficient representation for them. Currently, only an initial characterization has been made; we expect to finalize this effort in the future. The current characterization first separates finite difference (FD) grids and finite element (FE) meshes. FD grids (characterized by the implicit availability of adjacency information) are then classified as either orthogonal or non–orthogonal. For orthogonal grids, only the coordinate points in each dimension are maintained. For non–orthogonal grids, all the grid points are maintained in a linear ordering based on the natural (i.e., lexicographical) ordering of the points. FE meshes are currently maintained in the most general manner; i.e., as lists of nodes (i.e., points) and elements. Each element consists of some number of nodes with the first nodes defining the shape of the element. Adjacency information (node–node, node–element and element–element adjacencies) is maintained as lists also.

Geometry decompositions are a methodology for solving PDE problems in parallel. The global (continuous or discrete) geometry is partitioned into several subdomains and the computation at each domain is assigned to one (or more) processor(s) of a parallel machine. Since a large class of PDE solvers eventually require discrete representations, our representation concern is mainly for discrete partitionings. A discrete partitioning consists of identifying the set of nodes and elements that belong to each subdomain as well as the interaction between adjacent subdomains. The adjacency information generally consists of node–node, node–element and element–element adjacency data and may in fact be computed as needed. While there are two types of decompositions (node decompositions and element decompositions), the representations are exactly the same as only indirect references to nodes or elements of some mesh are maintained in the decomposition object. We expect to support continuous partitionings in the future.

3.6 Specification of PDE Solution Schemes

The solution of a PDE problem can be viewed as the application of a sequence of transformations where each step transforms the problem to a more tractable form until the desired final result is obtained. Each transformation produces a new object and also some associations which allow one to map the solution of the transformed problem to the solution of the original problem. The application of a complete package can therefore be viewed as a composite transformation that produces the final result in one (large) step. Transformations could potentially span different PDE solver packages. They can thus involve complex iterations between multiple packages where each package is executing in its own address space. While the sequence of transformations model can be applied for the entire solution algorithm used, we restrict ourselves in this work to relatively "high granularity" transformations. The small granularity transformations or "point methods" will be considered in the future. [4] In

[4]While such small granularity transformations are very important, they are not typically of use for application problem solving environment builder; their main use would be by PDE solver researchers

this section we identify the structure of a solution scheme specification by identifying the transformations and control operators that can be applied.

At a high level, the solution scheme specification can be as simple as requesting that a certain PDE solver be used to being as complex as iterative algorithms that, for example, iterate over different operators in adjacent domains using interface relaxation techniques. In the simplest case, the information necessary for a specific solver are the representations of all the objects required for that solver. This information must be "registered" with the PDESpec translator when the solver is installed in the system. Hence, when a specific solver is used, only the name of the solver and any solver–specific parameters must be indicated to invoke that solver. The algorithm translator will generate representations of all the objects in the form expected by that solver and then invoke it.

The more interesting cases are, of course, iterations and other complex control mechanisms, symbolic transformations of objects and using components of one package in conjunction with components of another package. We consider symbolic transformations separately (see Section 3.10).

As mentioned earlier, discrete representations of the domain are used during the solution of a PDE problem. While such discrete representations can be generated statically for some solution schemes, transformations that generate discrete geometries from continuous geometries must be available dynamically as well (for example, for adaptive schemes, for free boundary problems and for problems with time dependent domains). This is achieved by a transformation that converts a continuous domain to a discrete domain by invoking some domain discretizer with appropriate parameters.

Consider linear elliptic PDE operators. In this case, the possible transformations are:

- a discretization operation that would transform the PDE problem to a system of linear algebraic equations as well as a mapping of the solution of the algebraic problem to the solution of the differential problem,
- a linear equation reordering operation that would reorder the system of equation (to achieve better results during the solution phase) as well as the inverse mapping,
- a linear system preconditioning operation that would precondition the system of equations (to achieve better convergence during the solution phase),
- a linear solver operation that would solve the system of equations, and
- a composite operation that would transform the PDE problem directly to the solution.

to experiment with alternative solution techniques.

For linearized nonlinear operators and for transient operators being solved by time–stepping, the same transformations apply, but with one or more additional parameters in the equation indicating the "previous" solution(s). Section 3.10 explains this in more detail.

For nonlinear operators, the discretization operation produces nonlinear algebraic equations. Since generated nonlinear equations cannot be represented explicitly in non–symbolic languages (e.g., C & FORTRAN), typically the nonlinear algebraic equations are not directly represented. Instead, a function that computes the residual is called during the iterative nonlinear algebraic equation solver. In this case, the transformation approach would view this process as a single step transformation that produces the PDE solution from the nonlinear equations directly.

Other possible transformations include transforming transient boundary value problems into systems of ODE initial value problems, direct symbolic generation of nonlinear equations / Jacobians and many small granularity "point" operations.

Once various transformations have been identified, one needs control structures to allow users to compose the transformers in arbitrary ways. The standard iterators (*for* and *while*) and conditionals (*if*) are provided for this purpose. A set of predicate functions that perform various operations on the results of transformations are provided for use in boolean expressions.

3.7 Representing PDE Solutions

As the representation of the results of a PDE "solver" is based on the properties of the solver, the methodology for representing PDE solutions is solver specific. If the solver is a finite difference type solver, then it produces discrete values for the unknowns (and possibly its derivatives) at an *a priori* specified set of discrete points in the problem domain. Hence, to represent such a solution one needs the problem domain, the discrete domain used by the solver as well as the discrete solution values. In addition, it is common to apply an interpolation scheme to provide access to the solution at other points of the domain. Since the order of the interpolation scheme is dependent on the order of the solution, we must also record the type of interpolation that must be performed.

Finite element solvers use a basis function expansion for the unknown functions(s). Hence, the in order to make sense of the result, we need to know the basis used, the coefficients produced and of course the mesh used to compute the solution. In the FEM case, a separate interpolation function is not necessary as the basis functions in fact provide this facility.

Once representations for PDE solutions are available, PDESpec can support the needs of complex iterations including those that involve multiple packages and multiple solution representations.

3.8 Syntax and Semantics of PDESpec

In this section we outline the syntax and semantics of PDESpec. A rigorous definition of PDESpec is given elsewhere [We94].

3.8.1 Basic Syntax

In developing PDESpec's syntax, we have chosen to use the MACSYMA language syntax as a base. This allows us easy access to MACSYMA's symbolic manipulation facilities and also allows us to avoid developing yet another syntax for basic language facilities. MACSYMA's syntax for expressions is the standard infix notation with the usual precedence rules. For iterations, MACSYMA has one very general iteration construct which captures *for*, *while* and *repeat* loops. For conditionals, it has the standard *if–then–else* clause. MACSYMA uses a comma as a statement separator within compound statements and semicolon as a statement terminator outside of compound statements. Comments are enclosed in "/$\star$" and "$\star$/".

3.8.2 Overview of Syntax and Semantics

A PDESpec program consists of a sequence of PDE problem data objects and a PDE solution scheme specification object. The syntax of each object has the following form:

<objecttype> '(' <arg_1> ',' <arg_2> ',' ... <arg_n> ')'

where <objecttype> is the type of object and <arg_1> ... <arg_n> are the arguments of that object. Each argument (i.e., attribute of the object) is a keyed argument; i.e., each argument is of the form:

<key> '=' <value>

where <key> is the keyword that identifies the argument and <value> is the value of the argument.

The types of objects in PDESpec are: *equation*, *domain*, *boundary_conditions*, *initial_conditions*, *mesh*, *decomposition*, *algorithm*, *solve* and *solution*. The first four are the problem specification objects and the latter are the solution specification objects or the solution itself. The *algorithm* object is used to specify the sequence of transformations that are to be applied to some combination of *equation*, *domain*, *boundary_conditions* and *initial_conditions*. In this view, an *algorithm* object is essentially a subroutine definition. The *solve* object is the application of a predefined *algorithm* to a specific equation, domain, boundary conditions and initial conditions. The *solution* object is the result of executing a *solve* object.

Every object is given a symbolic name. To represent inter–object dependencies, each object has associated with it a dependency list where the types and names of the objects that this object depends on are listed. An attribute–value list called a

property list is also associated with each object. The property list is used to store related information about the objects that are derived during the lifetime of the object (e.g., whether an equation is linear or not).

As we mentioned in our discussion of the structure of these objects, the representations of parts of some objects is case–specific. For example, the nature of the information that describes a 2–dimensional domain in piecewise parametric form is completely different from information that describes a 3–dimensional domain built using CSG (constructive solid geometry). Hence, the language representation for such objects has a "type" attribute at a high level. The type attribute is then used to discriminate between the types of information stored in the object. A further bifurcation occurs in cases where a large amount of information is stored, for example in a mesh object. In this case, we allow the data to be stored elsewhere (in a file for example) and for the object to contain only the meta–data or the data that describes the contents of the file.

3.8.3 Example: The Algorithm Object

As an example of the syntax of PDESpec, consider the *algorithm* object. The semantics of the algorithm object's attributes are:

- parameters
  This attribute lists the parameters of the algorithm. The parameters of the algorithm are similar to those of a subroutine, using the function–algorithm analogy. Each parameters is typed and could be any of the valid types (i.e., a basic type or one of an object type or a structured type). Syntactically, the parameter list is written using MACSYMA's list delimiters '[' and ']' with comma separation. Each parameter specification has the syntax "*type* (*identifier*)", where *type* is the type of the parameter and *identifier* is the local identifier given to it.

- body
  The value of the body is the list of statements that comprise the body. Each statement is either a built–in MACSYMA construct such as an iterator or a conditional, a type declaration declaring temporary variables within the body of the algorithm, or the application of a transformation.

Consider the semantics of algorithm objects. As mentioned earlier, an algorithm object is essentially a subroutine definition; it only defines a set of actions to be taken when it is appropriately invoked. An algorithm is invoked when it is instantiated within a solve object.

The body of an algorithm consists of the sequence of statements to be executed. Withing the body, standard block–structured language support for nested blocks and definitions is provided using MACSYMA's language syntax and semantics. Each statement can be either a type declaration for a local variable, one of a set of valid MACSYMA statements or the invocation of a transformation supported by PDESpec.

Type declarations are done using a *declare* statement. Type declarations are necessary only when a local variable is needed, or when the type cannot be inferred or when the inferred type may be different from what is desired (e.g., in the case of single precision vs. double precision floating point numbers).

The set of accepted MACSYMA statements include the (general) iteration construct, the conditional statement and some miscellaneous statements such as return statements.

The transformations supported by PDESpec include domain discretization, domain partitioning, operator discretization (for systems of linear PDEs), linear system reordering, linear system preconditioning, linear system solution, changes of variables in operators, nonlinear PDE operator linearization, symbolic discretization of PDE operators (e.g., for time discretized iterations), mapping solutions on one discrete domain to another, and mapping solutions of a problem to another (e.g., mapping the solution of a linear system to the solution of an elliptic PDE problem) and combination transformations that perform composite transformations. Each transformation operator has a specific type signature which is used to validate the validity of the sequence of transformations applied in the body. When an algorithm object is compiled, some of the transformations are actually performed by the compiler (for example, the symbolic transformations) while others are converted into library calls.

Figure 3.1 shows an instance of the algorithm object. The algorithm specifies an ELLPACK elliptic problem solution algorithm using "Bi–Linear FEM" as the operator discretizer, "As is" indexing of the linear system (i.e., no reordering), linear system solution using the "Jacobi CG" iterative method and output of the computed solution, the true solution (which must be supplied by the caller, of course) as well as the error.

3.9 Implementation

Given a symbolic PDESpec program, the PDESpec compiler must generate code the invokes the appropriate packages to achieve the desired results. The compiler is implemented as a translator that translates a PDESpec program into an imperative language (e.g., C or FORTRAN) which is then compiled and linked with libraries. In this section, we address the issue of what needs to be generated and how it is generated. The run–time system of PDESpec is also considered.

3.9.1 Representing the PDE Problem Components

Each PDESpec problem specification object has in addition to its textual representation, several run–time representations that are made available to various PDE solver components. For example, the equation object has one representation which is used for the case where a linear operator is specified in a functional form for an ELLPACK operator discretization module. If the operator is given in a variational form and an

```
algorithm (
  name = "ellpack FEM solver",
  parameters = [
    equation (e),
    domain (d),
    boundary_conditions (b),
    mesh (m)
  ],
  body = [
    object (e),
    object (d),
    object (b),
    object (m),

    module ("ellpack", "discretization", "bi-linear FEM"),
    module ("ellpack", "indexing", "as is"),
    module ("ellpack", "solution", "jacobi CG"),
    module ("ellpack", "output", "solution"),
    module ("ellpack", "output", "true"),
    module ("ellpack", "output", "error")
  ],
  properties = [
    [Template, string, "Ellpack linear"],
    [System, string, "ellpack"]
  ]
)
```

Figure 3.1: An example algorithm object.

ELLPACK module is used, then additional (symbolic) transformations are automatically done by the compiler to generate a suitable representation. If a residual–type representation is needed by a solver, then that form is generated.

Thus, depending on the solver used, the representation of each object needed is different. In order to be extensible, each solver must be "installed" into PDESpec by specifying the desired representations for each object for each input object of that solver. The facilities that support the integration of existing (or "foreign") solvers (and other related components) is called the *foreign system interface* of PDESpec.

3.9.2 Representing Discrete Geometries

For discrete geometries, we again provide multiple representations that can be used depending on the needs of the particular solver. When solvers with differing needs are used, an expensive translation process is needed. Although expensive, this facilitates previously unavailable interaction between software components.

For most solvers, the discrete geometry information is static. That is, the discrete information is only read by the solver and not modified. However, for some solvers the discrete geometry varies dynamically during the solution process. In such cases, a more flexible (hence more complex) representation is used to facilitate the adaptive operation.

3.9.3 Translating Algorithms

Algorithm objects contain the sequences of statements that must be executed to achieve the desired results. Some of these statements are actually compile–time macros that need to be evaluated by the compiler. For example, the *linearize* macro to linearize nonlinear PDE operators is such a case. To linearize a nonlinear operator, one essentially performs a Freéchet series expansion of operator equation and truncates terms of second or higher order. This truncated series (containing the Fréchet derivative of the operator) is then reorganized to be in the desired form. This transformed equation is then used in some iterative solution process. All symbolic transformations that can be performed at compile time are done in a similar manner.

Most non–symbolic transformations expand to code sequences that are installed into PDESpec when that transformation is defined. When such a tranformation is used, it is expanded to the corresponding code sequence. These code sequences are then forwarded to the final program that is executed.

3.9.4 Execution

The run–time environment generated for algorithms depends on the particular combination of solution modules that are present in the algorithm. If the components allow one uniformly compatible set of representations, then a single address space execution environment is used. However, in cases where incompatible representations are simultaneously needed, a multi–address space execution with each incompatible

solution module in a separate address space is used. These address spaces are interconnected via the PDELab software bus, PDEBus. Interface relaxation algorithms involving the same solver on different domains and different operators is an example of this type of execution.

If some transformation in an algorithm needs run–time symbolic manipulation facilities, then that is achieved via the remote access facilities of PDEBus. Once the symbolic results are available, they are made accessible to the non–symbolic execution environment via various evaluation mechanisms.

Other execution options such as interactive execution control, run–time data gathering and computational steering are also implemented by multi–address space executions via PDEBus.

3.9.5 Representing Computed Solutions

Once a solution has been computed (where a "solution" is the result of a transformation that with an appropriate type signature), it is internally represented as discussed earlier in Section 3.7. This representation is then used later for updating iteration variables and other such initializations.

3.10 Symbolic Manipulation in PDESpec

In this section, we illustrate the various symbolic manipulation operations that take place in PDESpec. Several of the transformations performed by PDESpec are symbolic and also some of the automatic representation conversions performed by PDESpec are symbolic.

3.10.1 Symbolic Transformations

Consider the *linearize* transformation for symbolically linearizing nonlinear PDE operators for solution via Newton's method. Given a PDE operator of the form $F(u) \equiv 0$ with boundary conditions $G(u) \equiv 0$, the idea of the method is to approximate these with a first order Fréchet series expansion

$$\begin{aligned} 0 \equiv F(u) &\approx F(u_0) + F'(u_0)(u - u_0) \\ 0 \equiv G(u) &\approx G(u_0) + G'(u_0)(u - u_0) \end{aligned}$$

and then iteratively solve these linear problems starting with some initial guess u_0. $F'(u)$ and $G'(u)$ are the Fréchet derivatives of the operators F and G with respect the the function u and its derivatives. The corresponding symbolic/numeric process that implements Newton's method can be described as follows:

> Compute Frechet derivatives $L(u)$ and $B(u)$ of $F(u)$ and $G(u)$
> Initialize u_0

repeat

Solve $L(u_0)u = -(F(u_0) - L(u_0)u_0)$, $B(u_0)u = -(G(u_0) - B(u_0)u_0)$

Set $u_0 = u$

until converged.

The linearize macro in PDESpec generates the linearized operator equation L and boundary conditions B and then the user implements any desired iteration scheme around it.

Consider the operator discretization transformation for the case of discretizing a time dependent PDE operator. In this case, the time derivatives are replaced by difference quotients and then the resulting elliptic system is solved iteratively via time marching. For a parabolic PDE operator of the form $u_t = F(u)$, discretizing the PDE operator via the generalized Crank–Nicholson procedure produces:

$$\frac{u(t) - u(t - \Delta t)}{\Delta t} = \lambda F(u) \mid_{u=u(t)} + (1 - \lambda) F(u) \mid_{u=u(t-\Delta t)}$$

Note that we have suppressed the space variables and derivatives of u in the above equation. Assuming that the solution and its derivatives are known at the $t - \Delta t$ level, then the problem is solved by iterating over the time variable. The corresponding symbolic/numeric process that implements this time–stepping method can be described as follows:

Compute the generalized Crank–Nicholson discretization of $u_t = F(u)$

Set $t = t_0$

while $t \leq t_{end}$ do

Solve $\lambda F(u) \mid_{u=u(t)} - \frac{u(t)}{\Delta t} = (1 - \lambda) F(u) \mid_{u=u(t-\Delta t)} - \frac{u(t-\Delta t)}{\Delta t}$

Update t

end

3.10.2 Representation Translation and Code Generation

PDESpec also applies various symbolic manipulation tasks during its normal processing. We consider three such uses here.

As described earlier, a PDE operator can be described in one of many forms. Similarly, each PDE solver requires that the operator be specified in a specific representation, which of course may be different from the form that the use specified the operator in. In this case, PDESpec applies symbolic operators to translate from one representation to another. This involves differentiation, integration and coefficient extraction, for example.

If the operators are nonlinear and if an iteration is performed after linearing the operator, then the Jacobian of the PDE operators is necessary. PDESpec automatically generates the Jacobian in any desired representation by using the symbolic

differentiation capabilities of MACSYMA. In some cases, when user–defined functions are part of the PDE operator, automatic differentiation [BG92] is needed as well.

Once PDESpec has completed the manipulation of the program, it uses an automatic code generation tool to generate code from the symbolic representation it has of the program [WW92].

3.11 Examples

3.11.1 Steady–State Heat Flow in a Reactor

Consider a reactor with a steel dome and a concrete base as shown in Figure 3.2. The inside surface of the dome is initially 450°K and the ambient temperature around the reactor is 80°K. It is assumed that no heat is lost through the bottom surface of the dome or the base. We are trying to find the temperature T such that

$$\nabla \cdot k\nabla T = 0, x \in \Omega$$

Here k is the thermal conductivity, $k = 30.62\frac{cm^2}{min}$ in Ω_1 (steel) and $k = 0.79\frac{cm^2}{min}$ in Ω_2 (concrete).

$$\nabla T \cdot \mathbf{n} = 0$$

$$T = 450^0 F$$

$$-k\nabla T \cdot \mathbf{n} = 0.7(T - 80^0 F)$$

where $\mathbf{n}$ is the exterior unit outward normal to $\partial\Omega$.

Let us first define some default settings that assist us in the specification.

```
/* global defaults for all objects */
defaults (
    dimension = 2
);

/* defaults for all equation objects */
defaults[equation] (
    variables = [ x, y ],
    solutions = [ T ],
    depends = [ T(x,y) ],
    coordinates = cartesian,
    aliases = [ Dx(T) := diff (T, x),
                Dy(T) := diff (T, x),
                grad(T) := [Dx(T), Dy(T)],
    type = differential
);
```

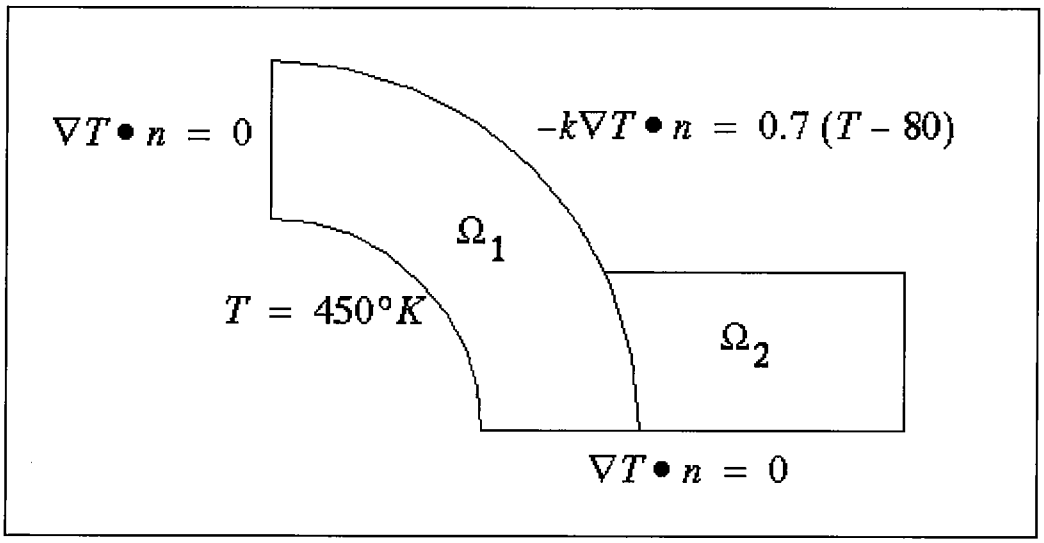

Figure 3.2: The domain of the reactor heat flow simulation. Ω_1 is the steel casing around the reactor and Ω_2 is the concrete support. While the entire reactor is the 3–dimensional domain created by rotating this domain around the left end, due to symmetry we need only apply the simulate on the shown slice.

The domain of the simulation is written in a piecewise parametric form as given below.

```
domain (
    name = "dome",
    type = piecewise_parametric,
    boundary = [
        orientation = clockwise,
        parametric (x=3, y=.7-t, t, 0, .7),
        parametric (x=3-t, y=0, t, 0, 2),
        parametric (x=cos(t), y=sin(t), t, 0, pi/2),
        parametric (x=0, y=1+t, t, 0, .75).
        parametric (x=1.75*cos(pi/2-t), y=1.75*sin(pi/2-t),
                                                t, 0, pi),
        parametric (x=1.6039015+t, y=.7, t, 0, 1.3960985) ]
);
```

The interior equation is defined below. The thermal conductivity $k(x, y)$ is an external function that must be defined appropriately over the entire domain.

```
equation (
    name = "steady-state heat flow",
    domain = "dome",
    expressions = [ Dx( k(x,y)*Tx ) + Dy( k(x,y)*Ty ) = 0 ],
```

```
    properties = [ [self-adjoint], [steady-state] ]
);
```

The boundary conditions are given below. The boundary conditions are also specified piecewise (as the domain) with each condition being applicable to the corresponding piece of the boundary of the domain.

```
/* the following three equations are defined in order to
   simplify the boundary conditions definition */

equation (
    name = "heat-flux",
    expressions = [ -k*grad(T).Tn = 0.7*(T-80) ]
);

equation (
    name = "insulated",
    expressions = [ grad(T).Tn = 0 ]
);

equation (
    name = "dirichlet",
    type = algebraic,
    expressions = [ T = 450 ]);

boundary_conditions (
    name = "bcond",
    domain = "dome",
    equation "steady-state heat flow",
    conditions = [ "heat-flux", "heat-flux", "heat-flux",
                   "insulated", "dirichlet", "insulated" ]
);
```

We have now completely specified the problem. The next object defines the mesh to be used in the finite element simulation of this problem.

```
mesh (
    name = "mesh",
    type = file,
    filename = "/tmp/reactor.mesh",
    filetype = pdelab
);
```

The following algorithm object defines an algorithm that can be used to solve any steady–state flow problem that ELLPACK can solve. The required input objects are specified as parameters.

```
algorithm (
    name = "ellpack steady-state flow solver",
    parameters = [equation(e), domain(d),
                  boundary_condition(bc), mesh(m) ],
    body = [
        e, d, bc, m,
        module ("ellpack", "discretization", "Bi-linear FEM"),
        module ("ellpack", "solution", "Jacobi SI", itmax = 100),
        module ("ellpack", "output", "T") ],
    properties = [
        [Template, string, "Ellpack linear"],
        [System, string, "ellpack"]
    ]
);
```

The following solve object applies the "ellpack steady-state flow solver" algorithm with specific input objects to solve the reactor heat flow problem.

```
solve (
    name = "solve",
    algorithm = "ellpack steady-state flow solver",
    parameters = [ "steady-state heat flow", "dome", "bcond",
                   "mesh" ],
    properties = [
        [System, string, "ellpack"]
    ]
);
```

Executing the solve object above will result in the solution of the simulation. Figure 3.3 shows the computed solution.

3.11.2 Modeling the Drying and Shrinkage of Salami

This example shows the problem description of a model of the drying and shrinkage of salami. From page 768 of [IK79] we quote:

> Salami is modelled by a hygroscopically covered homogeneous and isotropic equivalent colloidal material by interpreting the water concentration as a distributed parameter based on the dry mass. The sorption equilibrium of the salami is determined by the state of the casing in the given air–technical surroundings. Considering that the length to diameter ratios is generally > 6, end–effects have been neglected and the transport of heat and mass was assumed to occur in the radial direction only. It was also supposed that mass transfer in the rod occurs in liquid form and has no influence on the heat transfer inside the material. The heat capacity in the casing has been neglected.

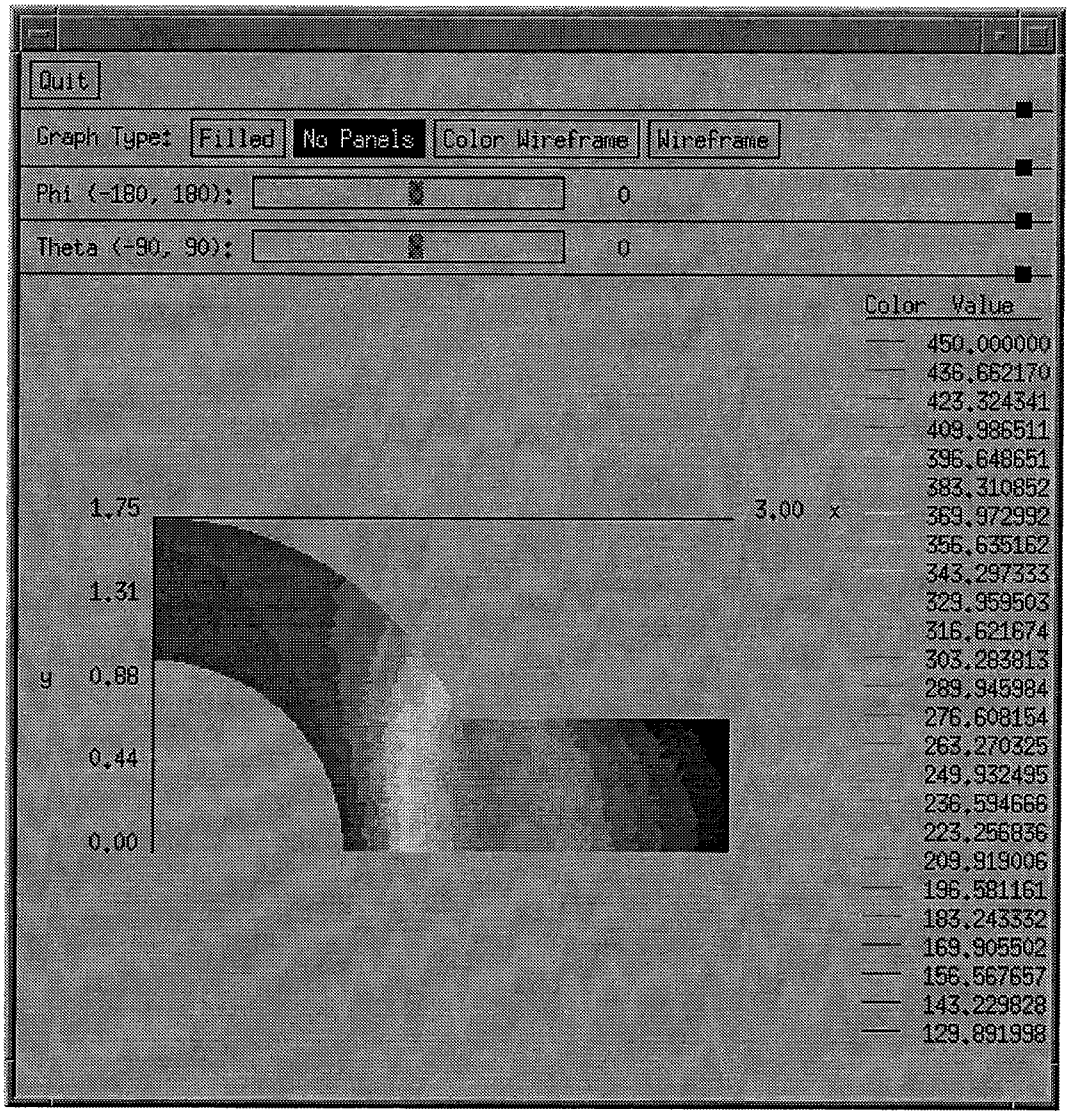

Figure 3.3: Solution of reactor simulation.

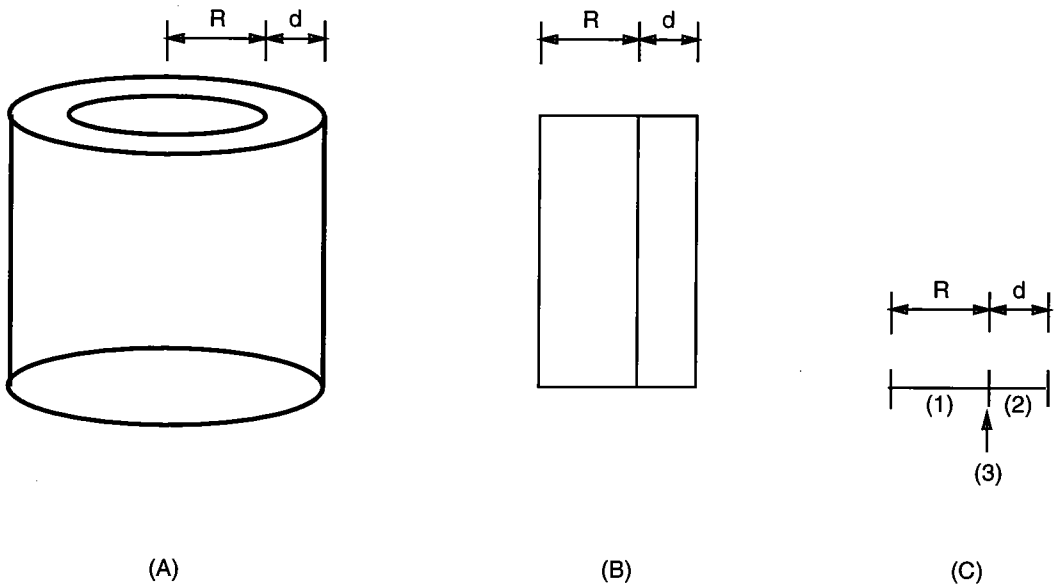

Figure 3.4: Modeling drying and shrinkage of salami. (A) shows the initial model with R being the radius of the "inside" of the salami and with d being the radius of the casing. (The figure greatly exaggerates d; generally d << R.) (B) shows the first abstract model arrived at by considering radial symmetry. Then, ignoring the corner effects (possible if the length of the salami rod is high), we can use axial symmetry to reduce the problem to the 1–dimensional model shown in (C). The three components of the domain are labeled (1), (2) and (3).

We show below the mass balance problem definition. The domains are expressed in PDESpec as follows:

```
/* domain (1) from the figure: the inner part */
inner : domain (
  dimension = 1,
  left = 0,
  right = R,
  parameters = [real(R)]
);

/* domain (2) from the figure: the casing */
outer : domain (
  dimension = 1,
  left = R,
  right = R+d,
  parameters = [real(R), real(d)]
);

/* domain (3) from the figure: interface point */
interface : domain (
  dimension = 1,
  left = R,
  right = R,
  parameters = [real(R)]
);

/* simulation time */
simtime : domain (
  dimension = 1,
  left = 0,
  right = T, /* T is the time limit of the simulation */
  parameters = [real(T)]
);
```

The mass balance equations are:

```
mb1 : equation (
  dimension = 1,
  coordinates = polar,
  variables = [t,r], /* r is the radius, t is time */
  solutions = [Xp],
  depends = [Xp(t,r)],
  type = differential,
```

```
  expressions = [
    diff(Xp,t) = Dp * (diff(Xp,r,2) + 1/r*diff(Xp,r))
  ],
  constants = [Dp = ?],
  domains = [[r, inner], [t, simtime]]
);

mb2 : equation (
  dimension = 1,
  coordinates = polar,
  variables = [t, r], /* r is the radius, t is time */
  solutions = [Xc],
  depends = [Xc(t,r)],
  type = differential,
  expressions = [
    diff(Xc,t) = Dc * diff(Xc,r,2)
  ],
  constants = [Dc = ?],
  domains = [[r, outer], [t, simtime]]
);

mb3 : equation (
  dimension = 1,
  coordinates = polar,
  variables = [r], /* r is the radius, t is time */
  solutions = [Xp, Xc],
  depends = [Xp(r,t)],
  type = differential,
  expressions = [
    Dp*rhop*diff(Xp,r) = Dc*rhoc*diff(Xc,r)
  ],
  constants = [Dp = ?, Dc = ?],
  parameters = [real(rhop), real(rhoc)],

);
```

The boundary and initial conditions for the mass balance equations are:

```
initial_conditions (
  domain = inner,
  equation = mb1, /* implicitly IC on inner region */
  conditions = [Xp(t=0, r) = Xp0],
  parameters = [real(Xp0)]
);
```

```
initial_conditions (
  domain = outer,
  equation = mb2, /* implicitly IC on outer region */
  conditions = [Xc(t=0,r) = Xc0],
  parameters = [real(Xc0)]
);

mb1bc : boundary_conditions (
  domain = inner,
  equation = mb1,
  conditions = [
    free,   /* free on left end of domain */
    -Dp*rhop*diff(Xp,r) = Kpc*rhop*(Xp-Xepc) /* right end */
  ],
  constants = [Dp = ?, Kpc = ?, Xepc = ?],
  parameters = [real(rhop)]
);

mb2bc : boundary_conditions (
  domain = outer,
  equation = mb2,
  conditions = [
    free, /* free on left end == interface between two regions */
    -Dc*rhoc*diff(Xc,r) = Kca*rhoc*(Xc-Xeca) /* right end */
  ],
  constants = [Dc = ?, Kca = ?, Xeca = ?],
  parameters = [real(rhoc)]
);
```

3.12 Conclusion

This chapter describes the rationale, design and partial implementation of PDESpec, a high level language for specifying partial differential equation problems and their solution schemes. We consider issues in representing PDE problem components, solution specifications and solutions and developed unifying representations for them. A summary of the syntax and semantics of PDESpec was given along with more detailed descriptions of algorithm objects. We also described the implementation in terms of the operations performed by the compiler and the run–time system. Finally, the symbolic manipulation done in PDESpec was highlighted and some examples given.

The PDESpec language is somewhat verbose and clumsy for human use. In fact, this language is meant as an "assembly language" for PDE computations. That is, this language is meant as the representation language for higher level tools and

(application specific) natural languages – a human user should not be expected to write PDESpec code directly.

Chapter 4

Computational Intelligence for Problem Solving Environments

The numerical solution of partial differential equation models depends on many factors including the nature of the operator, the mathematical behavior of its coefficients and the exact solution, the type of boundary and initial conditions, and the geometry of the space domains of definition. There are many proposed alternatives for the numerical solvers of such PDEs and mathematical software for some of them has already appeared. Moreover, each of the solvers is characterized by a number of parameters that the user must specify. In most cases the user is interested in a solution within a specified error level that satisfies certain resource (e.g., memory and time) constraints. This chapter addresses the problem of selecting a PDE solver and its parameters for a given PDE problem such that the numerical solution satisfies the user's computational objectives. We apply an expert system methodology based on the so–called *exemplar reasoning approach.* In this chapter we describe the PYTHIA expert system that implements this methodology within the domain of applicability of the parallel ELLPACK library [HRC+90] that includes the sequential ELLPACK [RB85] modules. PYTHIA's reasoning approach is based on the performance profiles of various ELLPACK solvers observed for a population of a priori determined elliptic PDE problems. These profiles are automatically generated out of known sets of performance data for each module of the library. PYTHIA's advice mode is realized in two phases. First, the characteristics of the current problem are matched with those of the existing PDE problem population to select the most relevant PDEs. Then the profiles of solvers for the selected class of problems is used to predict the two pairs, (grid, method) and (configuration, machine) that satisfy the user's objectives. PYTHIA's framework and methodology is general and applicable to any class of PDE problems and solvers.

4.1 Introduction

It has been predicted that future problem solving environments will include at least by some form of intelligence and will provide some "natural" user interface within well defined domains of applications [GHR92]. The purpose of this study is to address the issue of intelligence within a specific class of applications that can be described by a mathematical model involving PDEs defined on general geometric regions. The design objectives and architecture of such PSEs are described in [WHR+94b]. The goal of these PSEs is to assist the user to carry out the numerical solution of these models and visualize their solutions. Depending on the mathematical characteristics of the PDE models, there are "thousands" of numerical methods to apply, since very often each of the several phases of these methods are parameterized. On the other hand, the numerical solution must satisfy several objectives, primarily involving error and hardware resource requirements. It is unrealistic to expect that engineers and scientists will or should have the deep expertise to make "intelligent" selections of both methods and their parameters plus the selection of computational resources that will satisfy their computational objectives. Thus, we propose to build a computational advisor or expert system to make the "right", or at least "good" selections. Unfortunately, the knowledge about PDE solvers is vast. Fortunately, it is classified according to some general characteristics of the mathematical models.

The main objective of this work is to develop a paradigm of computational intelligence for selecting pairs (method, machine) and their parameters for the numerical solution of PDE models. For verification and demonstration purposes, we have selected the ELLPACK PDE library whose modules can be combined to define a large number of PDE solvers. Without loss of generality, the methodology is implemented for PDE problems solvable by the ELLPACK solvers. The applicability of these solvers is restricted to single equation PDE boundary value problems of the form

$$\begin{aligned} Lu &= f \quad \text{on} \quad \Omega, \\ Bu &= g \quad \text{on} \quad \partial\Omega \end{aligned} \tag{4.1}$$

where L is a second order linear elliptic operator, B is a differential operator involving up to first order partial derivatives of u, and Ω is a bounded open region in 3–dimensional space. The PYTHIA expert system is based on a database (DB) of performance data for each PDE solver using a set $\mathcal{H}$ of hardware platforms and for a priori defined parameterized population $\mathcal{P}$ of elliptic problems of type (4.1). The knowledge base (KB) of PYTHIA consists of a priori defined rules and facts together with a posteriori ones derived from the known data (DB, $\mathcal{H}$, $\mathcal{P}$) using the exemplar–based knowledge acquisition and learning approach [Bar86]. PYTHIA's inference strategy consists of matching the characteristics of the given PDE problems with those of one PDE problem P_0 in $\mathcal{P}$ or a subclass $\mathcal{S} \in \mathcal{P}$. Then, the known performance data for P_0 or $\mathcal{S}$ and the set of applicable solvers (U_a) are used to identify one (or several) solver and machine pairs that satisfies the user's objectives (e.g., percentage error and turn around time).

This chapter is organized as follows: Section 4.2 discusses some related work in solving the algorithm selection problem in the domain of PDE solvers. Section 4.3 presents an overview of the computational paradigm of PYTHIA. The PYTHIA knowledge bases are described in Section 4.4. The inference algorithm implemented to produce an advice is given in Section 4.5. Section 4.6 discusses the methodology used to handle uncertainty. Section 4.7 discusses an alternate approach to identifying problem classes using backpropagation–based neural networks. The overall PYTHIA architecture and its implementation is discussed in Section 4.8. Finally, in Section 4.9 we present the performance evaluation of PYTHIA for several scenarios.

4.2 Related Work

There have been several attempts at developing expert systems for assisting in various aspects of the PDE solution process. In [Ric76], Rice describes an abstract model for the algorithm selection problem. The algorithm selection problem is defined as the problem of determining a selection (or mapping) from the problem space (or its more accessible counterpart, the feature space) to the algorithm space. The "best" selection is then the mapping that is better (in the sense of having a better performance indicator in the performance measure space) than other possible mappings. Using this abstract model, in [Ric79] Rice describes an experimental methodology for applying the abstract model in the performance evaluation of numerical software.

In [MOF90], Moore et. al. describe a strategy for the automatic solution of PDEs at a different level. They are concerned with the problem of determining (automatically) a geometry discretization that leads to a solution guaranteed to be within a prescribed accuracy. The refinement process is guided by various "refinement indicators" and refinement is affected by one of three mesh enrichment strategies. At the other end of the PDE solution process, expert systems such as [KU90] can be used to guide the internals of a linear system solver. This particular expert system applies self–validating methods in an economical manner to systems of linear equations. Other expert systems that assist in the selection of an appropriate linear equation solver for a particular matrix are also currently being researched and developed.

In [DG89, DG92], Dyksen and Gritter describe an expert system for selecting solution methods for elliptic PDE problems based on problem characteristics. Problem characteristics are determined by textual parsing or with user interaction and are then used to select applicable solvers and to select the best solver. This work differs significantly from ours; our work is based on using performance data for relevant problems as the algorithm selection methodology whereas Dyksen and Gritter's is based solely on problem characteristics. We argue that using problem characteristics solely is not sufficient because the characterization of a problem includes many symbolically and *a priori* immeasurable quantities, and also because practical software performance depends not only on the algorithms used, but on the particular implementations of those algorithms as well.

In [KME93] Kamel et. al. describe an expert system called ODEXPERT for

Table 4.1: Examples of PDE problem characteristics for elliptic operators.

PDE problem characteristics			
Operator	Boundary Conditions	Functions	Domain
Poisson	Dirichlet	Smooth	Rectangle
Laplace	Neumann	Oscillatory	Square
Helmholtz	Mixed	Discontinuous	Non–rectangular
Self–adjoint	Homogeneous	Wave Front	Convex
Homogeneous	Linear	Singular	Nonconvex
Separable	Nonlinear	Peak	Geometric singularities
u_x, u_y, u_z			Union of domains
u_{xy}, u_{xz}, u_{yz}			Unit Square
Constant coefficients			Reentrant corners

selecting numerical solvers for initial value ordinary differential equation (ODE) systems. ODEXPERT uses textual parsing to determine some properties of the ODEs and performs some automatic tests (e.g., a stiffness test) to determine others. Once all the properties are known, it uses its knowledge base information about available ODE solution methods (represented as a set of rules) to recommend a certain method. After a method has been determined, it selects a particular implementation of that method based on other criteria and then generates (FORTRAN) source code that the user can use. If necessary, symbolic differentiation is used to generate code for the Jacobian as well.

4.3 An Overview of PYTHIA

In this section we present a formal definition of the selection problem that PYTHIA attempts to solve and the reasoning approach or computational intelligence paradigm utilized.

4.3.1 The PDE Solver Selection Problem

In a PSE for PDE based applications the user starts by defining the components of the corresponding mathematical problem. In the context of //ELLPACK this consists of specifying the model (4.1) and its domains Ω in some high level representation. In an ideal scenario the system extracts the type of the differential operator, boundary conditions and the mathematical behavior of its input (i.e., coefficients and right sides of the equations) and its output, the exact solution. Table 4.1 lists examples of characteristics for each component of the PDE problem. In the first implementation of PYTHIA most of this information is assumed to be provided by the user. The automatic generation of most of this information is still a research issue that we hope to address in the future. Moreover, we assume that the user requests to receive a solution that is within some error bound and produced within some time interval.

Two of the important design objectives of a PSE for PDE applications are first, to provide advice in selecting the mathematical model that describes "best" the application and, second, to select a (method, machine) pair to numerically solve the PDE. In this work we only address the second objective. Specifically, PYTHIA attempts to determine an optimal, or at least good, strategy for solving a PDE problem of type (4.1) under *accuracy* and *time* constraints imposed by the user. From among the applicable methods in library of PDE solvers, PYTHIA must identify pairs (grid, method) and (configuration, machine) that minimize the computation cost, that is, it determines the four "variables" so that

$$\min C\,(grid,\ method,\ configuration,\ machine)$$

is achieved subject to the constraints $\|u - u_h\| \leq \epsilon,\ t \leq \mathcal{T}$. Here u_h denotes the approximation of the exact solution u, ϵ and $\mathcal{T}$ are the specified accuracy and time constraints, respectively, and C is the computational cost measured, say, in some monetary terms. In this work we assume a fixed sequential machine with known configuration bounds. The case of heterogeneous hardware platforms will be addressed later.

4.3.2 PDE Problem Characteristics

It is clear from the previous discussion that the goal of finding the "best" (method, machine) pair for a given PDE problem and computational objective depends on the PDE problem's characteristics. This information is quantified in the PDE problem *characteristic vector*

$$\vec{C}_{PDE} = \left(\vec{O}, \vec{BC}, \vec{F}, \vec{D}\right)$$

where each of the subvectors consists of the, a priori defined, characteristics of each PDE problem component, see Table 4.1. The value (α) of each entry of the subvectors ranges in the interval [0,1] and indicates the presence level of each characteristic (e.g., $\alpha = 0$ means pure absence and $\alpha = 1$ pure presence). For type characteristics (for example, Laplace and Dirichlet), we assign binary values to the related entries of the characteristic vector. The type characteristics can be easily extracted by parsing the representation of (4.1) described in Chapters 2 and 3. This is done within the PSE environment (see //ELLPACK [HRC+90]). The identification of the mathematical behavior of the functions is a much harder problem. Depending on their assumed representation, this information can be extracted by symbolic, numeric and imagistics (computational vision) techniques. These techniques are under development and have not been implemented in the current version of PYTHIA. Instead, it is expected that the user provides some of this information in a question and answer session. The exact codification of the PDE problem characteristics vector in PYTHIA is illustrated in Section 4.4.1.

PROBLEM #28	$(w\,u_x)_x + (w\,u_y)_y = 1,$ where $w = \begin{cases} \alpha, & \text{if } 0 \le x, y \le 1 \\ 1, & \text{otherwise.} \end{cases}$
DOMAIN	$[-1,1] \times [-1,1]$
BC	$u = 0$
TRUE	unknown
OPERATOR	Self–adjoint, discontinuous coefficients
RIGHT SIDE	Constant
BOUNDARY CONDITIONS	Dirichlet, homogeneous
SOLUTION	Approximate solutions given for α = 1, 10, 100. Strong wave fronts for $\alpha \gg 1$.
PARAMETER	α adjusts size of discontinuity in operator coefficients which introduces large, sharp jumps in solution.

Figure 4.1: A problem from the PDE population.

4.3.3 Population of Elliptic PDEs

The first step of PYTHIA's reasoning approach is the identification of an a priori defined PDE problem or class of problems whose characteristic vector is "close" to that of the problem specified by the user. The success of this approach thus relies greatly on having available a reasonably large population of various PDE problems about which some performance information is known. For the class of linear second order elliptic PDEs we are currently using the population created in [RHD81] for use in the evaluation of numerical methods and software for solving PDEs. It consists of fifty–six linear, two–dimensional elliptic PDEs defined on rectangular domains. Forty–two of the problems are parametrized which leads to an actual problem space of more than two–hundred and fifty PDEs. Many of the PDEs were artificially created so as to exhibit various mathematical behaviors of interest; the others are taken from "real world" problems in various ways. The population has been structured by introducing measures of complexity of the operator, boundary conditions, solution and problem. Figure 4.1 shows an example problem from the PDE population. The actual implementation of this population is reported in [BRH79].

The population also contains information about the properties of the problems. This information is encoded as a bit–string and is accessible via the ELLPACK[RB85] Performance Evaluation System (PES) [BRH79]. These properties are transferred to the PYTHIA characteristic vector representation by an automated procedure explained later in Section 4.8. Characteristics not identified in the ELLPACK PES are identified by the characteristic extraction component of PYTHIA.

We have already seen that PYTHIA's reasoning approach requires the identification of similar classes of problems. The search for similar problems is reduced by first comparing the given problem characteristic vector $\vec{C}$ with vectors that represent some common classes of PDEs. We define the characteristic vector of a class as the

element–by–element average of the characteristic vectors of all class members. That is, the i–th element of the characteristic vector of problem class c is defined to be:

$$(\vec{C}\,(c))_i = \frac{1}{|c|}\sum_{p\in c}(\vec{C}\,(p))_i.$$

Class identification and performance evaluation of PDE solvers is part of the knowledge acquisition for PYTHIA. The class identification is based on identifying problems from $\mathcal{P}$ whose $\vec{C}_i$ range within a specified subinterval of [0,1]. Once a class is identified the following information is generated and stored in the KB: i) the list of member problems, ii) the average characteristic vector, and iii) applicable solvers. The representation of problem features as a vector allows us to compare the characteristics of two classes (a single problem can be viewed as a class with one element) by measuring the distance between the two average vectors in some norm.

4.3.4 Performance Knowledge for Elliptic PDE Solvers

For PYTHIA to infer the "best" solution path for a given PDE problem, it needs to know the ranking of the applicable methods within the class of problems that have characteristics similar to the user's problem. This ordering (ranking) is a function of the user's computational objectives (accuracy and time levels). The generation of this ordering is done by applying statistical techniques to the performance curves or profiles such as *accuracy vs. degrees of freedom (dofs)*, [1] *dofs vs. time*, and *accuracy vs. time* for each (method, problem) pair. These profiles are generated by linear least squares approximation to raw performance data. The above three types of performance profiles are currently generated by the performance evaluation system (PES) system [BRH79]. For example, Figure 4.2 shows the raw performance data and Figure 4.3 displays the *dofs vs. time* profiles for seven //ELLPACK solvers on a problem from the PDE population $\mathcal{P}$. Finally, in Figure 4.4 we present a ranking of the methods in Figure 4.3 with respect to *dofs vs. time* profiles for a certain class and for the accuracy level of 0.05%.

4.4 The PYTHIA Knowledge Bases

The PYTHIA knowledge bases consist of 1) a priori rules specified by a (human) expert, 2) facts, and 3) rules generated from the performance knowledge for the PDE solvers considered.

[1] The number of degrees of freedom that are used in the discretized equation is the number of linear equations that are present in the system of equations produced by the operator discretizer. For most discretization schemes this is approximately the same as the number of discrete points in the domain.

Problem 10
Parameter Set 2 ($a = 50.0, b = 0.5$)
Method b4p2unix/1/14/46/

nx	ny	h	$nunk$	$rerr$	$rerrmax$	$errl2$	$resmax$
5	5	$0.25E+00$	9	$0.78E+00$	$0.80E+00$	$0.16E+00$	$0.15E+02$
9	9	$0.12E+00$	49	$0.23E+00$	$0.24E+00$	$0.16E-01$	$0.84E+01$
17	17	$0.62E-01$	225	$0.53E-01$	$0.58E-01$	$0.61E-02$	$0.57E+01$
33	33	$0.31E-01$	961	$0.13E-01$	$0.97E-02$	$0.31E-02$	$0.30E+01$
65	65	$0.16E-01$	3969	$0.32E-02$	$0.24E-02$	$0.16E-02$	$0.15E+01$

$resmxr$	$resl2$	$solmax$	nit	mem	tt	$t1$	$t2$	$t3$
$0.26E+04$	$0.12E+02$	$0.28E+00$	0	387	0.02	0.00	0.02	0.00
$0.32E+03$	$0.12E+02$	$0.81E-01$	0	2079	0.00	0.00	0.00	0.00
$0.22E+02$	$0.13E+02$	$0.66E-01$	0	14391	0.15	0.03	0.00	0.12
$0.15E+02$	$0.14E+02$	$0.63E-01$	0	106983	1.55	0.12	0.00	1.43
$0.26E+02$	$0.14E+02$	$0.63E-01$	0	822087	18.73	0.50	0.02	18.22

Figure 4.2: A sample of raw performance data generated by the performance analyzer. These numbers are for solving problem # 10 with parameter set # 2 ($a = 50.0$, and $b = 0.5$) using the "method" b4p2unix/1/14/46 with 5×5, 9×9, 17×17, 33×33, and 65×65 grids. The method string encodes the machine, operating system and solver (here 5–point star discretization, as–is indexing, and band Gaussian elimination).

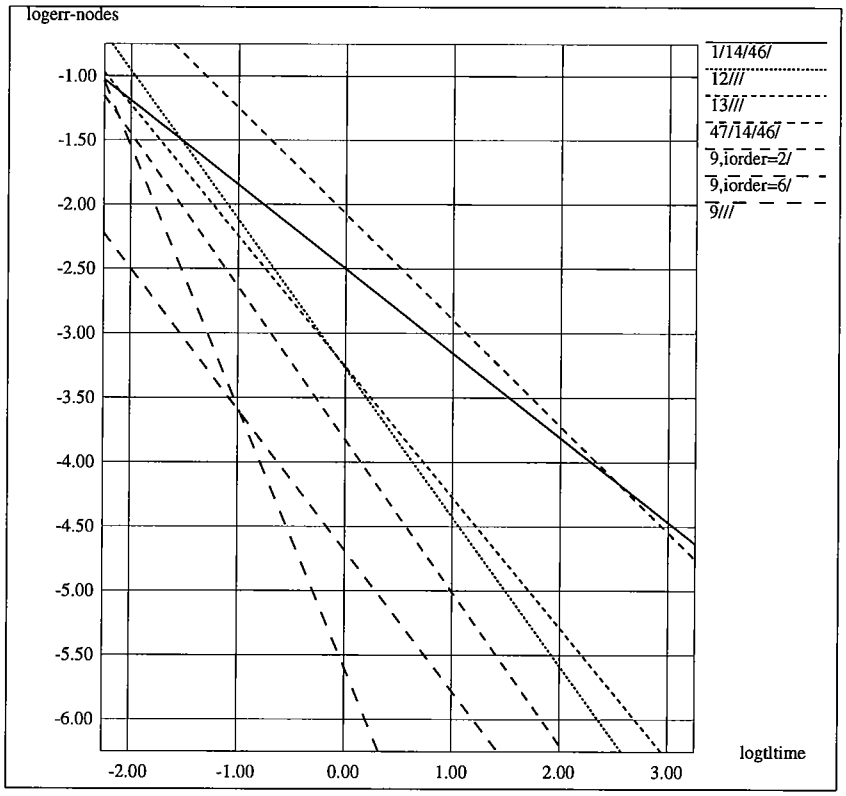

Figure 4.3: Performance profiles generated using the knowledge engineering environment of PYTHIA for a single PDE problem. The strings at the upper right are coded identifications of 7 solvers.

method	rank	min.	1st quart	median	3rd quart	max.
b4p2unix/1/14/46/	5.36	2.2E+01	5.1E+01	1.5E+02	2.1E+02	1.7E+03
b4p2unix/12///	5.86	2.0E+01	5.7E+01	2.1E+02	1.7E+03	1.0E+30
b4p2unix/13///	4.27	1.2E+01	4.4E+01	6.0E+01	1.5E+02	1.0E+30
b4p2unix/47/14/46/	1.91	2.4E+00	8.4E+00	2.0E+01	3.2E+01	1.0E+30
b4p2unix/9,iorder=2///	4.86	2.0E+01	5.0E+01	1.4E+02	1.8E+02	1.2E+03
b4p2unix/9,iorder=6///	2.68	4.9E+00	1.1E+01	1.8E+01	7.5E+01	1.0E+30
b4p2unix/9///	3.05	7.3E+00	1.3E+01	2.7E+01	3.6E+01	1.6E+02

Figure 4.4: A raw ranking table generated by the stochastic analyzer. This is for a 0.05% error level with the *dofs vs. time* as the performance criteria.

4.4.1 Facts

The knowledge about each problem in the population $\mathcal{P}$ and the various identified subclasses is referred to as *facts* of the PYTHIA knowledge base. Recall that many solvers are created by composing various modules in the //ELLPACK library so we also have facts about their compatibility. In addition, we have facts about the types of PDE problems to which solvers can be applied. This information allows PYTHIA to automatically form legal PDE solvers out of software parts. In the case of elliptic solvers, we have four types of modules [RB85]: discretization, indexing, solution and triple.

Problem Facts

This information is the characteristic vector of each PDE problem. For example, the vector

$$\left(\underbrace{2}_{1}, \underbrace{0,0,0,1,0,0,0,0,1,0,1,0,0}_{2}, \underbrace{0.0,0.0}_{3} \right)$$

represents the characteristics of the PDE operator of the problem specified in Figure 4.1 with the parameter $\alpha = 10.0$.

The first number indicates that the operator is two dimensional. The second set of numbers indicate that the operator is not Poisson, not Laplace, not Helmholtz, is self–adjoint, does not have constant coefficients, does not have single derivatives, does not have mixed derivatives, is not homogeneous, is linear, is not nonlinear, is elliptic, is not parabolic and is not hyperbolic, respectively. The last two numbers are the subjective measures of the smoothness and local variation properties of the operator. In this study, we have assumed only linear PDE problems and solvers. Usually solving nonlinear problems is reduced symbolically to solving a sequence of linear ones [WHR92]. In these cases, PYTHIA can be applied to select the appropriate solver for the linearized PDE problem.

Class Facts

This information consists of the set of member identifiers and the average characteristic vector of the class as defined in Section 4.3.3. This vector represents the centroid of all the member problems.

4.4.2 Rules

This information is characterized by some level of uncertainty or degree of confidence and it is usually derived by some approximate analysis [HRCV88] or provided by an expert. In order to control the level of uncertainty, PYTHIA also develops rules for specified classes of problems and applicable methods from the performance knowledge acquired through actual runs of methods for each member of the class. This approach has the advantage that the learning phase is implemented by just updating

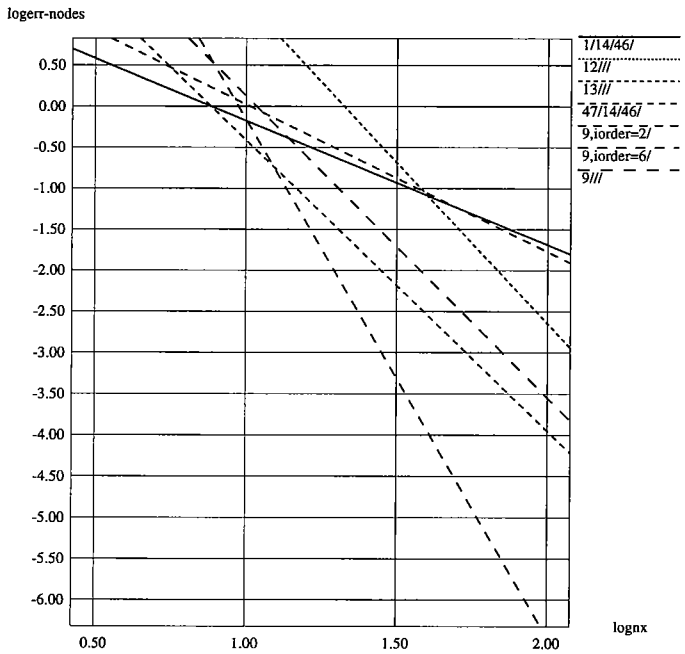

Figure 4.5: Knowledge base rules showing relative error as a function of the number of nodes (dofs = nx) in the discretized domain. The methods are identified (in coded form) at the upper right.

the performance knowledge either by adding more problems or by including more methods. This section describes the derivation procedure of PYTHIA rules and their representation. A priori defined rules are included using the same representation.

Problem Rules

This information is comprised of the performance profiles for each (method, problem) pair. As we have seen, they are straight line approximations to data points whose coordinates are the values of two performance indicators. The pair of indicators considered in each case is referred to as the performance criteria. The general form of this type of rule is:

For method m on problem p and performance criteria i,
the performance profile is (slope, intercept)

The performance criteria used so far include (accuracy, time) and (accuracy, dofs). In these two cases, the generated rules are of the form:

For method m on problem p, accuracy = f(n)
For method m on problem p, accuracy = g(t)

where the functions f and g are the linear profiles extracted from the experimental data for the (p, m) tuple. Figures 4.5 and 4.6 show examples of such rules.

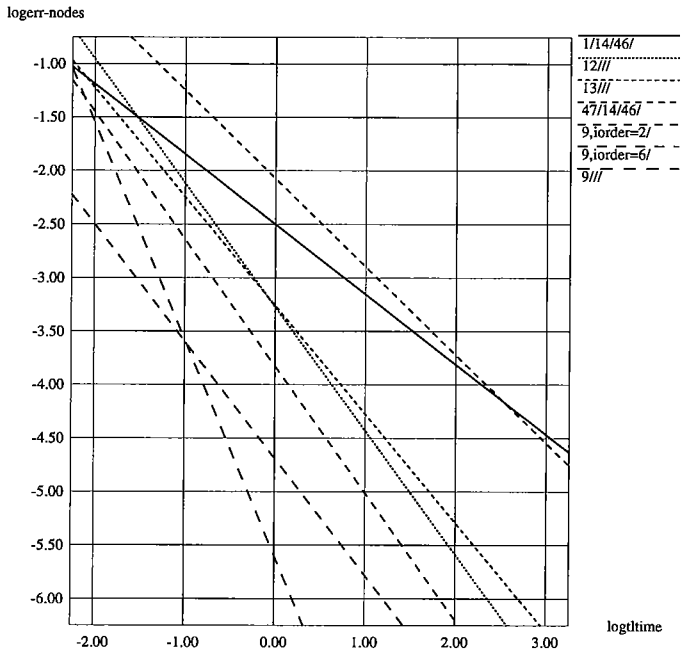

Figure 4.6: Knowledge base rules showing relative error as a function of the time taken to achieve that error. The methods are identified (in coded form) at the upper right.

Class Rules

This information provides the average performance of the method for a class of problems. These rules are automatically generated by PYTHIA by applying a stochastic methodology on problem rules or method profiles. To increase the degree of confidence for these rules one has to use classes with larger numbers of members and consider multiple performance criteria. The rule is encoded in the form of a step function where each step is defined by a preset level of (accuracy, time) computational constraints. The general form of this rule is:

For class c and performance criteria i,
the method ranked n–th at error level e is m with confidence c_1 *and* c_2

The same performance criteria as with problem rules are used here. The two confidence numbers are generated by the PYTHIA stochastic analyzer and indicate the confidence in the ranking. The first number, c_1, indicates the confidence that the method ranked m-th is in fact better than the method ranked $(m+1)$-th and c_2 indicates the confidence that the method ranked m-th is better than method ranked n-th, where n is the total number of methods applicable to that class of problems. Figure 4.7 indicates an example of a class rules for a class of problems from the population in [RHD81].

```
(class-perfdata (name HR.3-10) (rank 1) (method "47/14/46")
   (criteria nx) (errlevel 5pc) (conf1 0.80) (conf2 1.00))

(class-perfdata (name HR.3-10) (rank 2) (method "9,iorder=6//")
   (criteria nx) (errlevel 5pc) (conf1 0.80) (conf2 1.00))

(class-perfdata (name HR.3-10) (rank 3) (method "9//")
   (criteria nx) (errlevel 5pc) (conf1 0.80) (conf2 0.99))
```

Figure 4.7: Some class rules that rank the performance of various methods for problem class "HR.3–10" based on the number of nodes (NX) in the discretized domain to achieve a 5% relative error level.

User–Specified Rules

This information consists of assumptions about the performance of methods for various classes of problems. They are products of a priori and a posteriori analyses or observations made by experts. In some instances these rules have general acceptance while in many others these rules are subject to opinion. We plan to incorporate rules of this type that are the results of some a posteriori analysis [HR82, DRR88].

4.5 The Inference Algorithm

There are two significant parts of PYTHIA: the learning process and the selection task or the inference algorithm. In this section we address the latter. Specifically, we describe the process of combining the facts and rules in the knowledge base to provide a selection (grid, method) pair for the user's problem subject to specified constraints. The input to this phase of the reasoning process includes i) *problem features in the form of a characteristic vector*, ii) *user's computational objectives* (accuracy in % and time interval), iii) *relative emphasis of achieving the subobjectives of (ii)* (e.g., 70% emphasis on achieving the accuracy requirement and 30% emphasis on achieving the time requirement), iv) *weights for the criteria used in the rules*, and v) *the norm for computing the distance* $d(p, \mathcal{S})$ *and* $d(p, q)$ *with* $p, q \in \mathcal{P}$ *and* $S \subset \mathcal{P}$.

The inferencing logic in PYTHIA is organized as a Bayesian belief propagation network [Pea88]. Figure 4.8 shows a pictorial view of the PYTHIA reasoning framework. A detailed discussion of the techniques used for handling uncertainty is given in the next section. This section describes the overall algorithm without explaining in detail the logic used to compute the various numbers involved.

As mentioned earlier, the goal of the inference algorithm is the determination of the best matching class, the best solver and the closest exemplar which is to be used as a basis for predictions. The inference algorithm is shown in Figure 4.9. $P(C = c)$

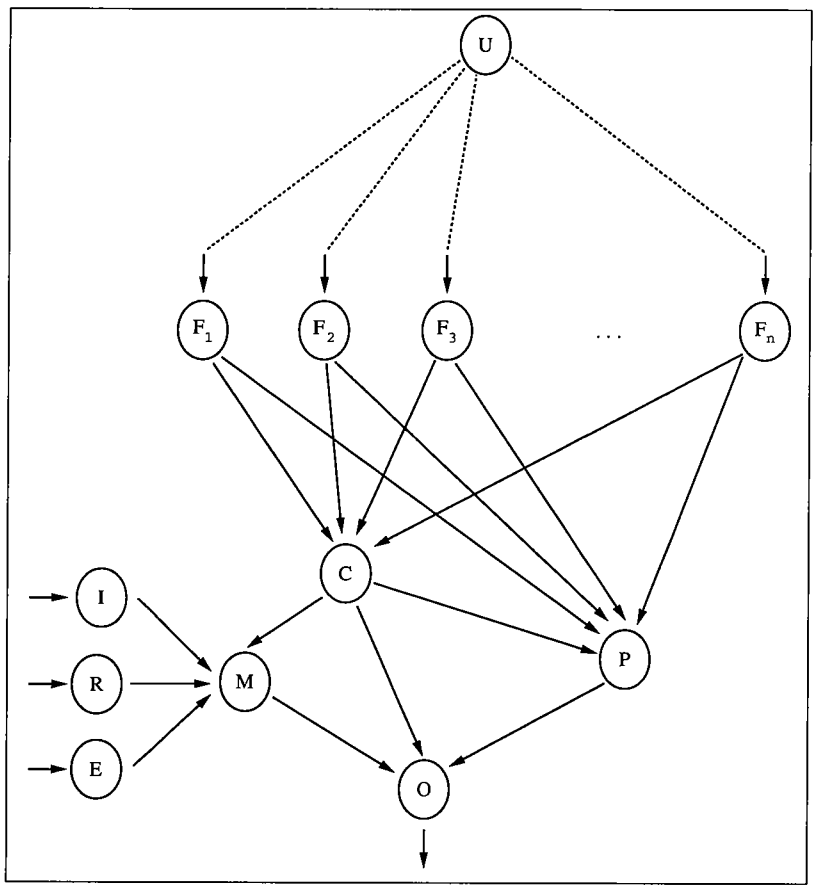

Figure 4.8: The reasoning framework of PYTHIA represented as a Bayesian belief network. The U node indicates the user's problem or the problem for which a solution is being sought. Nodes $F_1, \ldots, F_n$ represent the features of U. Node C represents the computation for determining the distance to various problem classes from U and node P represents the computation for determining a close exemplar for a given class. Nodes I, R and E indicate how the user wishes to weight the various method performance rules when determining the best method for a given class. Node M represents the computation to determine the best method to solve the problem and finally node O represents the computation done to suggest an appropriate method and parameters to solve the problem U.

- Compute $P(C = c)$ and sort
- Select best class c^0 in the $P(C = c)$ ordering
  Compute $P(M = m|C = c^0)$ using given rule weights and sort
  Compute $P(P = p|C = c^0)$ and sort
  Select best method m^0 in the $P(M = m|C = c^0)$ ordering
    Select closest problem p^0 in the $P(P = p|C = c^0)$ ordering
      Predict using (c^0, m^0, p^0) if possible, else backtrack
    If not possible to predict with m^0, then backtrack
  End
  If not possible to predict with c^0, then backtrack
End

Figure 4.9: The PYTHIA inference algorithm.

means the probability that the closest matching class is c. The best class c^0 is defined to be the class appearing first in the $P(C = c)$ ordering sorted in decreasing order. $P(M = m|C = c)$ means the probability that the best method from the set of methods applicable to class c is m and similarly, $P(P = p|C = c)$ means the probability that the best matching exemplar from class c is p. Thus, once these probabilities are calculated using Bayesian belief propagation rules, the algorithm basically applies a backtracking search to find the tuple (c^0, m^0, p^0) so that c^0 is the closest class, m^0 is the best applicable method and p^0 is the closest exemplar while satisfying the condition that the selected method be applicable to the user's problem.

Once the closest class c^0, the best applicable method m^0, and closest exemplar p^0 are known, we need to provide information about the number of dofs that should be used to satisfy the relative error requirement (i.e., how to discretize the domain) and to estimate how long that computation will execute. From the performance profile rules, we get profiles $f(n)$ and $g(t)$ for solving problem p^0 in class c^0 using method m^0, where

- n is the number of nodes in the discrete domain and $f(n)$ is the error in the solution computed using n nodes, and
- t is execution time and $g(t)$ is the error in the solution when the solution is computed in t time.

Since the amount of relative error in the computed solution and the amount of time taken to compute the solution are obviously coupled, we may or may not be able to completely satisfy the user's performance request that a certain accuracy be achieved within a specified amount of time. Hence, we may have to compromise. Figure 4.10 shows an example $f(n)$ and Figure 4.11 shows an example $g(t)$.

Suppose that the user requests that the error be less than e, the time be less than

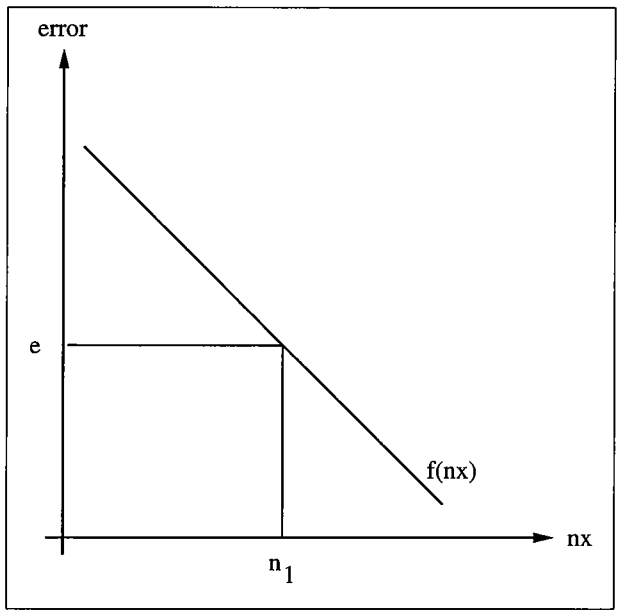

Figure 4.10: A performance profile for a solution method m on a problem p that relates the grid size n_1 to the error e achieved.

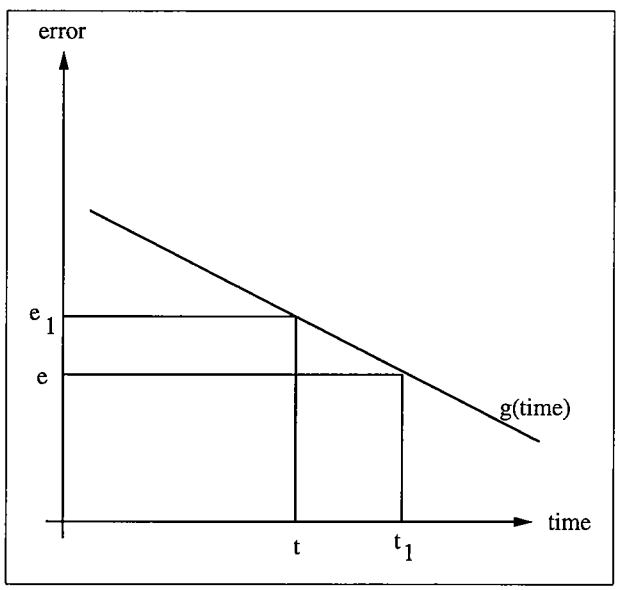

Figure 4.11: A performance profile for a solution method m on a problem p that relates the grid size (n_1 from the previous figure) to the time t_1 taken to solve the problem. The pair (t, e_1) illustrates Case 1 in the text where requested time t is smaller than the time required to meet the accuracy $= e$ request.

t and that the weight on the error request being satisfied is α. Assume that:

$$\begin{aligned} e_1 &= g(t) \\ t_1 &= g^{-1}(e) \end{aligned}$$

Then, we have the following cases:

- Case 1: $e_1 \leq e$
  If $e_1 \leq e$, then $t_1 \leq t$. Thus, we can satisfy the conditions that the time be less than t and that the error be less than e. The selection proposed is: *Use $f^{-1}(e)$ nodes to get a solution with a relative error of at most e in at most $g^{-1}(e)$ time.*

- Case 2: $e_1 > e$
  In this case, it is not possible to satisfy the conditions that the solution time be less than t and that the error be less than e. Hence, we find an intermediate point between (t, e_1) and (t_1, e) taking into account the user–specified weight for the relative importance of time and error. Hence, let

  $$(t^*, e^*) = \alpha(t_1, e) + (1 - \alpha)(t, e_1).$$

  Thus, the selection proposed is: *Use $f^{-1}(e^*)$ nodes to get a solution with a relative error of at most e^* in at most t^* time.*

4.6 Dealing with Uncertainty

PYTHIA's inference algorithm needs $P(C = c)$, $P(M = m|C = c)$ and $P(P = p|C = c)$, where $P(x)$ is the probability of the event x happening. As before, $P(C = c)$ means the probability that the closest matching class is c. $P(M = m|C = c)$ means the probability that the best method from the set of methods applicable to class c is m and similarly, $P(P = p|C = c)$ means the probability that the best matching exemplar from class c is p. In this section, we describe how the various problem and class facts and rules are used to compute these probabilities. We refer the reader to Figure 4.8 for the location of the nodes referred to below.

4.6.1 Evidence Nodes $F_1, \ldots, F_n$: Problem Features

Each of these evidence nodes represents one problem characteristic or feature with belief in a feature is represented as a probability. Boolean valued features use the discrete probabilities 0.0 and 1.0. For example, if an operator is known to be homogeneous, then the feature indicating homogeneity is assigned a probability of 1.0, representing complete knowledge in the homogeneity of the operator. If it is known to be non–homogeneous, then the feature is assigned a probability of 0.0. However, if neither case is known, then a probability of 0.5 is assigned to represent ignorance or equal belief in all the possibilities.

4.6.2 Node C: Finding Closest Classes

The computation performed at Node C is the calculation of probability distribution for all the classes, $c \in C$. That is, we want to compute $P(C = c|F_1 = f_1, \ldots, F_n = f_n)$.

Let $\overline{c}$ be the characteristic vector of class c. Compute $d = \|\overline{c} - \overline{u}\|$, where $\overline{u} = (f_1, f_2, \ldots f_n)$. Then, the marginal probability $P(C = c)$ is given by

$$P(C = c) = \alpha \left(1 - \frac{d}{\|\overline{d}\|_\infty}\right),$$

where $\overline{d}$ is the vector of all d's for $c \in C$, $\|\cdot\|$ is an appropriate norm and α is chosen so that $\sum_{c \in C} P(C = c) = 1.0$. This calculation therefore assigns the greatest probability to the class that is closest to the user's problem, as measured by the vector norm $\|\cdot\|$.

4.6.3 Node P: Finding Closest Problems

The computation performed at Node P is the calculation of probability distribution for all the problems in the PDE population. That is, we want to compute $P(P = p|C = c, F_1 = f_1, \ldots, F_n = f_n)$.

Let $\overline{p}$ be the characteristic vector of problem p. Compute $d = \|\overline{p} - \overline{u}\|$. Then, the marginal probability $P(P = p|C = c)$ is given by

$$P(P = p|C = c) = \begin{cases} \alpha \left(1 - \frac{d}{\|\overline{d}\|_\infty}\right) & \text{if } p \in c \\ 0 & \text{otherwise} \end{cases},$$

where $\overline{d}$ is the vector of all d's for $p \in P$ and α is chosen so that $\sum_{p \in P} P(P = p|C = c) = 1.0$. This calculation therefore assigns the greatest probability to those problems in each class c that are closest to the user's problem; that is, it identifies the most interesting exemplar from a class.

4.6.4 Evidence Node I: Performance Criteria Weights

This allows the user to specify the amount of weight to be placed on each performance criteria when selecting the solution method. Valid performance criteria are error and time. To specify that a weight of α is to be placed on the rules based on the "error" criterion, one gives $P(I = error) = \alpha$ and $P(I = time) = 1 - \alpha$.

4.6.5 Evidence Node R: Method Rank Weight

This allows the user to specify the amount of weight to be placed on performance profiles of each rank when selecting the solution method. Valid ranks are 1, ..., n, where n is the number of comparative rankings available. To specify that only those rules related to methods that ranked first in solving a particular class of problems are to be considered, one gives $P(R = 1) = 1$, $P(R = 2) = 0$, ..., $P(R = n) = 0$. Similarly, for equal weight on each rule irrespective of the rank, one gives $P(R = i) = \frac{1}{n}$, for $i = 1, \ldots, n$.

4.6.6 Evidence Node E: Error Level Weights

This allows the user to specify the amount of weight to be placed on performance data at each error level when selecting the solution method. Valid error levels are e_1, $\ldots$, e_n, where n is the number of error levels at which performance rules are available. To specify that only the rules relating to the first error level are to be considered, one gives $P(E = e_1) = 1$, $P(E = e_2) = 0, \ldots, P(E = e_n) = 0$. Similarly, for equal weight on each error level, one gives $P(E = e_i) = \frac{1}{n}$, for $i = 1, \ldots, n$.

4.6.7 Node M: Determining the Best Method

Here, we want to compute the belief in method $m \in M$ being the method that best satisfies all given conditions. That is, we want to compute: $P(M = m|C = c, I = i, R = r, E = e)$.

The knowledge base has facts that give for each class, the r–ranked method for a fixed performance criteria and error level along with two confidence numbers (say γ_1 and γ_2, respectively). The first number indicates the confidence that the specified method is better than the method ranked immediately after this one while the second number indicates the confidence that the method is better than the method ranked last. Hence, we let

$$P(M = m|C = c, I = i, R = r, E = e) = \gamma_1(c, i, r, e) \times \gamma_2(c, i, r, e).$$

However, for a given method m and class c, it is possible that there is no rule in the database that relates m and c. In this case, we let $P(M = m|C = c, \ldots) = 0$. (This is reasonable as such a rule not appearing in the database means either that the method is not applicable to that class of problems or that it performs very poorly in that class.)

The output of this node is the marginal probability of M. This is computed as follows:

$$P(M = m) = \sum_{c \in C, i \in I, r \in R, e \in E} P(C = c)\, P(I = i)\, P(R = r)\, P(E = e)\, \gamma_1(c, i, r, e)\, \gamma_2(c, i, r, e)$$

We also need $P(M = m|C = c)$. This is computed as follows:

$$P(M = m|C = c) = \sum_{i \in I, r \in R, e \in E} P(I = i)\, P(R = r)\, P(E = e)\, \gamma_1(c, i, r, e)\, \gamma_2(c, i, r, e)$$

4.6.8 Node O: Generating Output

The function of this node is to generate the final output; i.e., recommend a certain method as the best method to use to satisfy the given conditions and also indicate what size grid to use to achieve the specified error level within the given time bound. The reasoning used in this node is that of the previous section and is repeated here.

4.7 Using Neural Networks

As described in the previous section, we use a probability–based scheme to identify the best matching class for a given problem. We *a priori* partition the database of problems into a collection of problem classes using some criteria such as common characteristics. Then, each such class is represented by a single characteristic vector which is used in the class identification phase of the inference algorithm.

In this section, we describe the use of backpropagation based neural networks to implement this phase. Going back to the original problem, we are given sample vectors $v \in \mathcal{R}^n$ with each one representing a problem. We have to use these to divide $\mathcal{R}^n$ into m class zones C_i, $i = 1$ to m so that given another vector $u \in \mathcal{R}^n$, we can decide which class it belongs to. The situation is complicated by the fact that a given problem can belong to more than one class. In other words, two (or more) classes may overlap.

An alternate view of the problem is to consider it as a mapping problem. Let us suppose that we represent the m classes by a vector of size m. We treat each of its elements as a binary decision variable, with a 1 in the i^{th} position of the vector indicating membership in the i^{th} class. Our problem now becomes one of mapping the characteristic vector of size n into the output vector which shows class memberships.

To perform this mapping, we use a backpropagation based neural network. A backpropagation based network is essentially a supervised learning system consisting of an input layer, an output layer and one or more hidden layers. Each layer consists of a number of neurons. Such networks are feed–forward, and neurons in one layer connect to the neurons in the succeeding layer only. The connection can be complete, in which case each neuron in layer i connects to each neuron in layer $i+1$. Alternately, the connection pattern can be restricted and defined by the user. Each connection between two neurons has a strength associated with it, which is called its weight. The neurons are units performing extremely simple computation, and are highly abstract representations of their biological counterparts. Each neuron has a state (s_i), which is simply the weighted sum of all its inputs. The activation of the neuron, which is also its output (o_i), is simply some squashing function applied to the input. Mathematically

$$s_i = \sum_j w_{j,i} o_j$$

$$o_i = \frac{1}{1 + e^{-s_i}}$$

where $w_{j,i}$ is the strength of the connection between neuron j (of the previous layer) and neuron i. Initially, the weights $w_{j,i}$ are all assigned small random values. Then, the network is presented with the input. The output produced by the network is compared with the desired output and error values computed for all the output units. Using the backpropagation algorithm, these error values are used to compute error values for the hidden units. Weights are then changed in a way so as to reduce this error. This is essentially doing gradient descent on the error surface with respect to

the weight values. This cycle is repeated until the error falls below some predefined measure. See [RM86] for more details.

As we have mentioned earlier, we have reduced our problem to one of pattern matching. Backpropagation has been successfully used to solve similar pattern matching problems. In this case, our input consists of the characteristic vector, which has 32 elements [RHD81]. The output consists of a vector with 5 elements, corresponding to the number of classes that we use in our simulations and therefore to the number of classes that we wish to categorize the data into. Unlike the original non–neural heuristic, we are not imposing an arbitrary structure on classes by insisting that their central value lies at the mean of the samples that we have available. Rather we are letting the network discover the structure of the classes.

Since the input and output of the network are fixed by the problem, the only layer whose size is to be determined is the hidden layer. In the experiments described later, we arbitrarily chose this to have 10 elements. Also, since we have no *a priori* information on how the various input characteristics affect the classification, we chose not to impose any structure on the connection patterns in the network. Our network is thus *fully connected*, that is, each element in layer i was connected to each element in layer $i + 1$. Considering the small size of our network, this means the the number of connections is still small. To be precise, there are only $32 \times 10 + 10 \times 5 = 370$ connections in the network.

4.8 Architecture and Implementation

PYTHIA is organized as two components, the *knowledge engineering* and the *consultation environments*, connected via the various knowledge bases. These components exist as tools in the //ELLPACK system and are therefore accessible through //ELLPACK for general use. Both components of PYTHIA are implemented using CLIPS[Gia91], an expert system development shell developed by NASA.

4.8.1 CLIPS: *C Language Integrated Production System*

CLIPS is an expert system development shell implemented in C. CLIPS was originally only a forward chaining rule language based on the Rete algorithm (hence the production system part of the CLIPS acronym) [CW88]. However, it now supports two more programming paradigms: Procedural programming and object–oriented programming. The object–oriented programming language provided within CLIPS is called the CLIPS Object–Oriented Language (COOL). CLIPS comes in three forms: A system with a regular dumb–terminal interface, a system with a graphical user interface (for X Windows, MS–DOS, and MacIntoshes) or as an embeddable kernel system. The last form is extremely useful as it allows one to seamlessly integrate CLIPS into other programs without forcing users to learn about CLIPS. In fact, the CLIPS interfaces are all built by developing an interface around the CLIPS kernel.

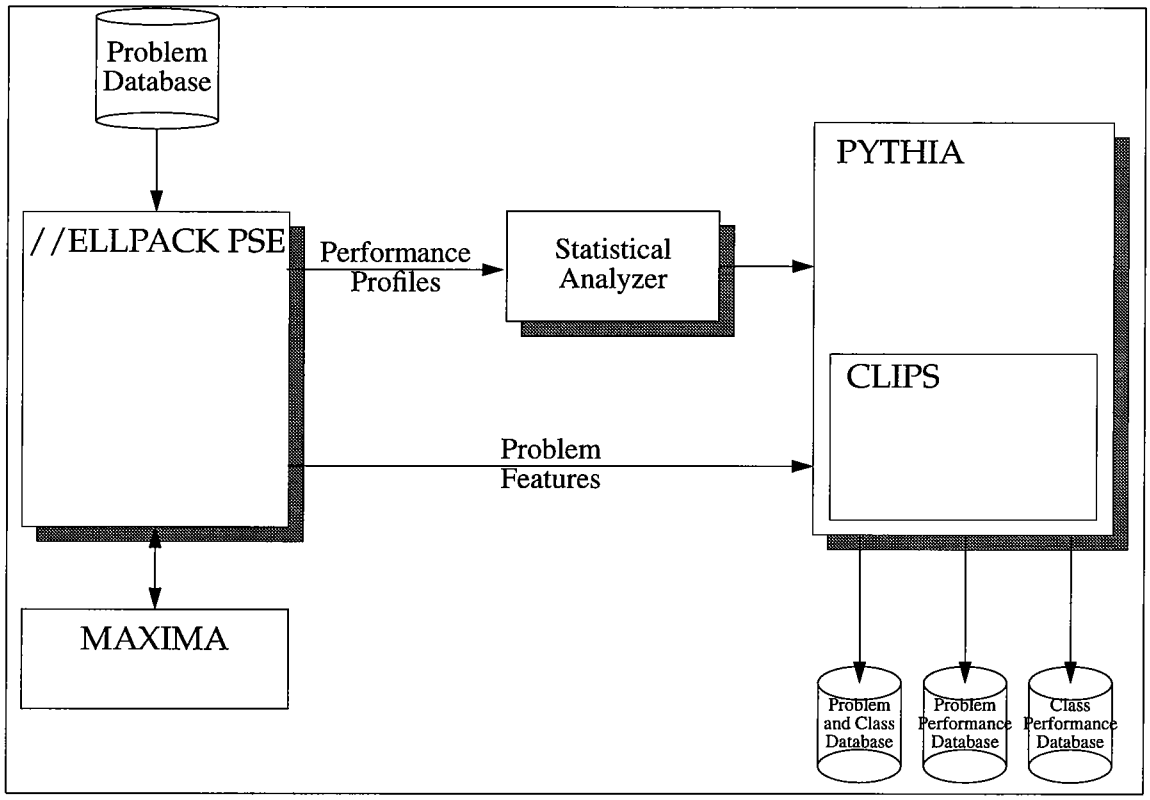

Figure 4.12: The architecture of the PYTHIA knowledge engineering environment.

We used the X interface during development of the CLIPS component of PYTHIA and then the embeddable version.

4.8.2 The Knowledge Engineering Environment

The PYTHIA knowledge engineering environment supports the development and maintenance of the PYTHIA knowledge bases described in Section 4.4. Figure 4.12 shows the architecture of this component. The knowledge engineering environment is implemented using a variety of languages/systems– CLIPS, FORTRAN, C, Perl, and shell scripts.

The ELLPACK problem database that we use consists of a set of structured files that are accessed using various manipulation programs. For performance data generation, we have developed convenient tools that allow the user to select a problem from the database, generate multiple test cases by specifying various solution methods and parameters (such as the number of grid lines), execute test cases, and finally extract relevant and interesting performance data. Once the performance data is available, it is run through a statistical analyzer whose task is to produce both linear least squares approximations to the performance data and a comparative ranking of the performance of all the methods applied to all the PDEs used in the test. The data produced by the statistical analyzer is automatically translated into CLIPS facts for

later use in the consultation environment.

The problems can optionally be run through the problem feature extraction tools that we have developed for MAXIMA[2] in order to determine their characteristics, as required by PYTHIA. Once all the necessary features are known (either by automatic extraction or by user specification), the features are also automatically translated into CLIPS objects for later use. Problem features are represented by instances of a "Problem" class that we have defined using CLIPS' object–oriented capabilities. The characteristic vector of a problem is computed by one of the methods defined for the "Problem" class by encoding all the properties in the form of a vector.

Problem classes (i.e., a collection of problems identified by some common properties) are represented in CLIPS as instances of a "Problem–Class" class that we have defined. A problem class object consists of a list of member problems (i.e., the names of instances of "Problem" objects) and a method that computes the characteristic vector of the entire class by averaging the sum of the characteristic vectors of all the class members.

Once all the problems and problem classes are defined, we generate some facts and rules for later use in the consultation environment: Facts describing problems, facts describing classes, rules describing the performance of various methods on problems, and, finally, rules describing the performance of various methods on classes of problems. All these facts and rules are implemented as ordered facts in CLIPS. These four knowledge bases are the basis on which the consultation environment makes its selections.

4.8.3 The Consultation Environment

The task of the PYTHIA consultation environment is to use the information generated by the knowledge engineering environment as a basis for making selections for users' about what method and parameters to use to solve a particular problem within specified performance constraints. The consultation environment is also implemented using a variety of languages/systems– CLIPS, C, Tcl, Perl, and shell scripts.

All the knowledge bases are loaded into PYTHIA when the consultation environment starts up. Let us assume that the user has specified a PDE problem in the //ELLPACK problem solving environment. Now, the user wishes to get some advice on what method to use and what the associated parameters are. At this point, //ELLPACK attempts to determine all automatically determinable features or characteristics of the problem by sending the problem description to the feature analyzer in MAXIMA. //ELLPACK then sends the problem description and all the known features to PYTHIA using the communication facilities of //ELLPACK. PYTHIA allows the user to specify some performance objectives and also what weight is to be placed on achieving each objective. Any unknown problem features are requested from the user at this time as well.

[2]MAXIMA is the Austin Kyoto Common Lisp version of the well known computer algebra system MACSYMA.

Once all this information is available, it is put into CLIPS' working memory via instance creations (for problem features) and fact assertions (for the performance objectives and rule weights). Firing CLIPS then executes the inference engine to compute a selection (a prediction as to what the best solver is). The selection and prediction data (the proposed method and its parameters as well as information on what accuracy is to be expected and how long the computation will take) is then propagated back to the problem solving environment. //ELLPACK examines the data and incorporates the corresponding modules into the solution algorithm being developed.

4.9 Performance Evaluation of PYTHIA

In this section we evaluate PYTHIA in terms of the "accuracy" of its predictions and its overhead. We evaluate the class selection process (as implemented in the neural network approach), the correctness of the selected method and parameters for various classes (in terms of the appropriateness of the suggested method as well as the correctness of the performance predictions) and finally the overhead of PYTHIA itself. All the performance tests use the ELLPACK PDE population described earlier.

4.9.1 Class Selection Using Neural Networks

To study the effectiveness of the class identification process using neural networks, we define the following non–exclusive classes (the number in parenthesis indicates the number of problems that belong to that class):

1. SOLUTION–SINGULAR: Problems whose solution has at least one singularity (6).
2. SOLUTION–ANALYTIC: Problems whose solution is analytic (35).
3. SOLUTION–OSCILLATORY: Problems whose solution oscillates (34).
4. SOLUTION–BOUNDARY–LAYER: Problems whose solutions depict a boundary layer (32).
5. BOUNDARY–CONDITIONS–MIXED: Problems with mixed boundary conditions (74).

There are a total of 167 problems in the population that belong to at least one of these classes. We split this data into two parts– one part contains two thirds of the exemplars and is used to train the network. The other part is used to test the performance of the trained network. All the simulations were performed using the Stuttgart Neural Network Simulator [ZMH+93], a very useful public domain simulator with a convenient graphical interface.

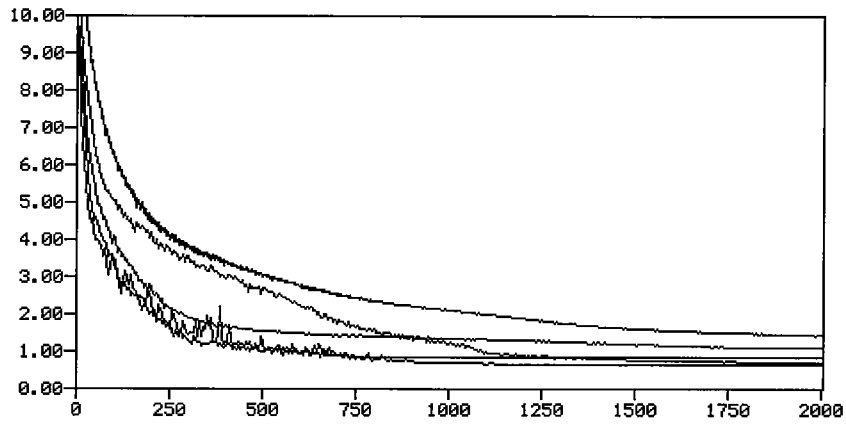

Figure 4.13: Plot of sum of squared error vs. iteration number for the first training algorithm.

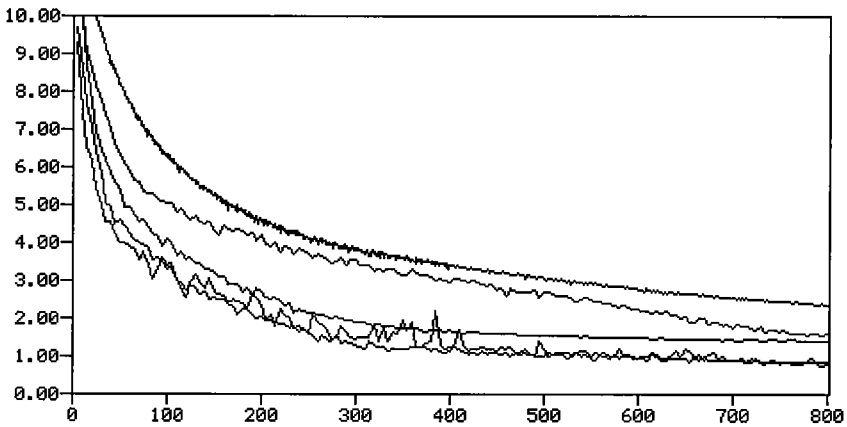

Figure 4.14: Magnification of the first part of Figure 4.14.

The first training algorithm used is a "vanilla" backpropagation routine. The *learning rate* is an important free parameter here. It is the value by which the gradient term is multiplied before being used to update the weights of the network. In Figure 4.13, we show the Sum of Squared Error (SSE) for the network as it changes with the training epochs (iterations). As expected, the best performance is obtained by some reasonably small value for this parameter, in this case 0.2. Using smaller values also leads to the same SSE value, but the learning takes much longer. Using larger values give oscillatory behavior, and the algorithm may get trapped in local minimum with a larger SSE value, as can be seen in the graph. Figure 4.14 concentrates on the early stages of iteration using various learning rate parameters to illustrate the behavior of the learning scheme there.

A popular variation on the standard backpropagation is to use backpropagation with momentum and flat spot elimination. This involves using another term to modify the weights, which is the second derivative of the error function. Also, a small constant

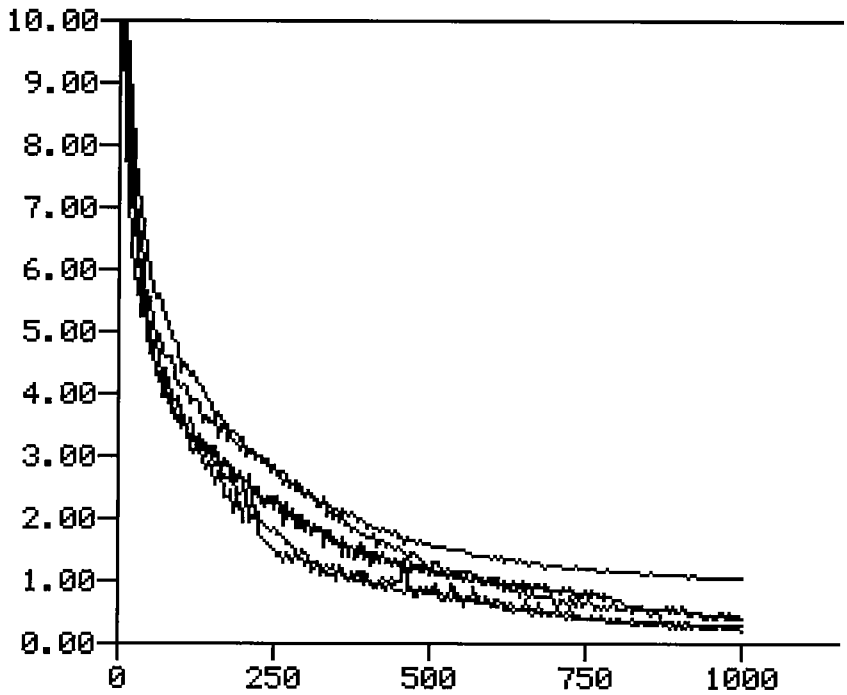

Figure 4.15: Plot of SSE vs. iteration number for the final training algorithm.

term is added to the derivative to avoid local minima. This is the second training algorithm used. The value of the learning rate was fixed at 0.2 from the previous experiments, and the flat spot elimination constant was chosen as 0.05. The parameter multiplying the momentum term was varied and once again plots of SSE vs. iteration number were made. The best performance seems to be for the momentum term at about 0.8. It was also clear that the addition of the momentum and flat spot elimination lead, as expected, to lower SSE values than the standard backpropagation, and to much faster convergence. In Figure 4.15, we show a plot of this algorithm with learning rate 0.2, flat spot elimination constant 0.05, and momentum multiplier 0.8.

Clearly the network learns to correctly classify all the problems it is trained on. However, one of the reasons we wished to use neural networks is that they could generalize. The next step therefore is to verify that our network was generalizing correctly. To do this, we split the total of 167 vectors that we have into two sets, one set consisting of 111 vectors (the "larger set"), and the other consisting of 56 vectors (the "smaller set"). First, the network is trained for 2000 epochs with the larger set. The learning rate used is 0.2, the momentum 0.8 and the flat spot elimination constant 0.05. After training, the network is presented with all the 167 vectors, and its output recorded. To interpret and analyze the output, we compute the least square norm of the expected and actual outputs for each of the 167 cases. Figure 4.16 shows a scatter plot of the results, with the X axis showing the vector number (from 1 through 167), and the Y axis showing the L_2 error norm. As can be clearly seen, the

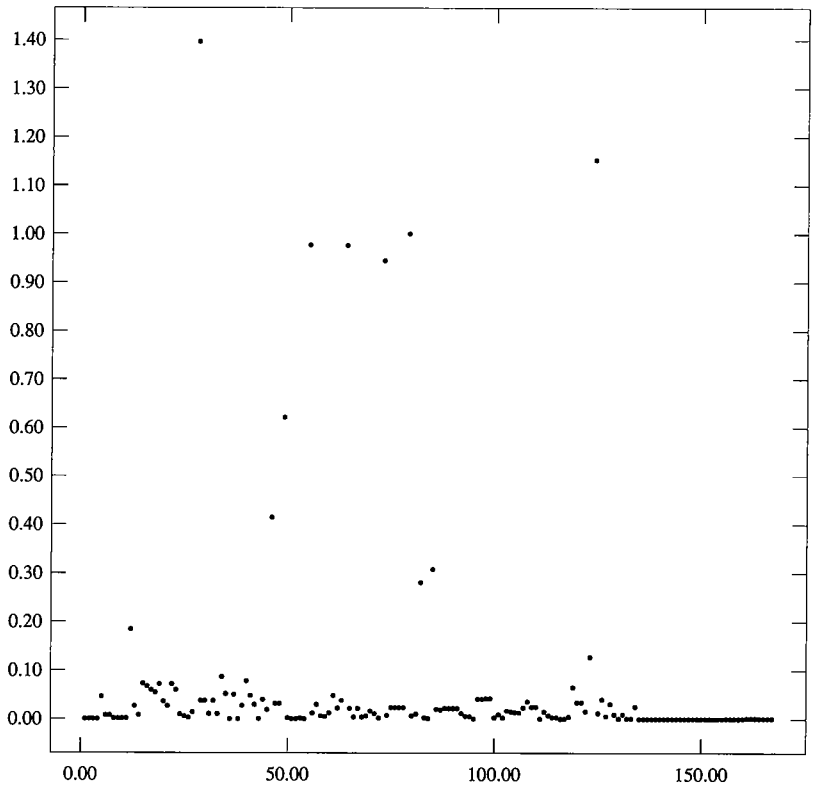

Figure 4.16: Scatter plot of error vs. vector number for the larger training set with the final training algorithm.

network correctly classifies all but a few of the vectors, which show up as outliers.

To further study the generalization power of such networks, we use the smaller problem set as a training set instead of the larger set as in the previous experiment. The parameters and the number of epochs remain the same as in the previous case and the testing is once again done with the complete set of vectors. The scatter diagram for this is shown in Figure 4.17. Again, we noticed that the number of outliers are few. However, as expected, training with fewer samples leads to a degradation of performance with respect to the previous case.

Tables 4.2 and 4.3 present some further analysis of this data. We first chose an arbitrary threshold for the L2 error norm. Error norms above the threshold would imply that the corresponding input had been misclassified. In Table 4.2, we show the number of correctly classified vectors (out of 167) for different values of the threshold. We can clearly see that the network is extremely successful in classifying correctly, especially with the larger training set. The same trend is reflected in Table 4.3, where we present the mean and median values for the error norms in the two cases. Notice that while the mean value of error for the smaller training set is slightly higher, the median value remains small. This clearly illustrates that while there are outliers,

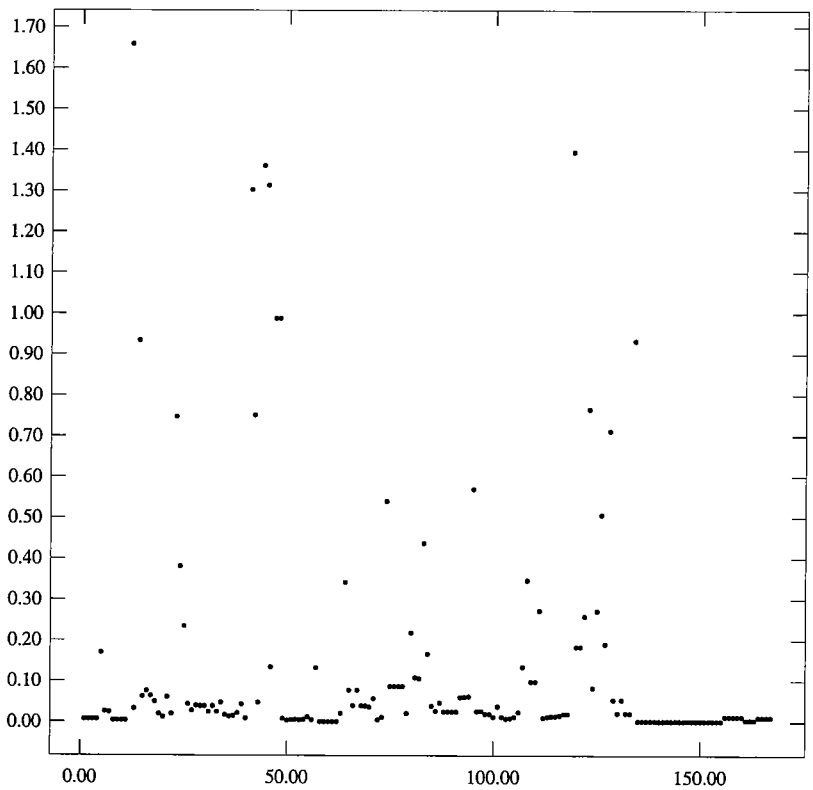

Figure 4.17: Scatter plot of error vs. vector number for the smaller training set with the final training algorithm.

Table 4.2: Number of correctly classified vectors with various training sets.

Training Set	Threshold Value			
	0.2	0.1	0.05	0.005
Large	157	155	143	67
Small	142	132	113	37

Table 4.3: Statistics for the error norms with various training sets.

Training Set	Mean	Median
Large	0.0652	0.0095
Small	0.1367	0.0235

most of the problem do get classified correctly.

4.9.2 Method Selection and Parameter Prediction

This experiment studies the effectiveness of the method selection and performance prediction methodology once a problem is identified as being in a certain class. We used the class of problems studied in [HR82] as the test class which we denote by c. Then, we use different parts of c as a "training set"[3] to make predictions for the rest of c.

In the first test, we use $\frac{c}{4}$ as a training set. We request selections to achieve relative errors less than 10^{-3}, 10^{-4}, and 10^{-5} in at most 10 minutes with equal weight on achieving the error bound and on achieving the time bound. For rule weights, we selected the default values chosen to emphasize the overall performance of methods (vs. peak performance) in making decisions. The results in this test are not very satisfactory: Only 15% of the selections were actually valid. (A selection is considered "invalid" if the suggested method is not applicable to the problem or if the any of the parameters do not apply correctly to the method.) However, it is encouraging that all the valid selection are in fact correct– solving the problems with the selected method and parameters does result in solutions with the requested error within the requested time. It should be noted that since equal weight is placed on the correctness of the error bound and on the correctness of the time bound, it is not necessary for both predictions (or either prediction) to fall within the originally requested bound for the selection to be considered correct.

In the second test, we used $\frac{c}{2}$ as a training set and specified the same error and time bounds as well as the same rule weights. The results here are much more satisfying:

[3]By training set we mean that the rules and facts that PYTHIA uses are ones derived solely from problems belonging to the training set.

All the selections made are in fact valid and 90% of them are also correct (i.e., the requested error is achieved within the requested time).

4.9.3 The Overhead of PYTHIA

The response time for a prediction from PYTHIA is generally in the order of seconds. We estimate the sizes of the various knowledge bases that PYTHIA uses to judge how much processing is done in this time.

For each problem, we have one instance of a "problem" class during the training phase and one ordered fact during the consultation phase. Since only one phase is active at a time, there is one object per problem. Let the number of problems in the database be np. For each problem p, there are $nmp_p \times nc$ performance data facts, where nmp_p is the number of methods with which p can be solved and nc is the number of criteria with which the performance of a method is measured. Thus, there are a total of

$$np + \sum_{p \in P} nmp_p \times nc$$

problem–related facts in the database.

Each class is represented by an instance of a "problem-class" class during the training phase. During the consultation phase, a class is represented by an ordered fact indicating the members, an ordered fact indicating the class characteristic vector and the class performance facts described earlier. Let nc be the number of classes. For each class c, there are $nr \times nmc_c \times ne \times nc$ class performance facts, where nr is the number of comparative ranks for which performance comparisons are available, nmc_c is the number of methods with which all the problems in class c have been solved, ne is the number of error levels at which performance data was gathered and nc is as before. Thus, there are a total of

$$2 \times nc + \sum_{c \in C} nmc_c \times nr \times nc \times ne$$

class–related facts in the database (where nmc_c is the number of methods with which all problems in class c have been solved).

For the ELLPACK PDE population described earlier, $np \approx 275$, $nc \approx 5$ (for the data we have generated), $nmp_p \approx 15$ (although 15 or more possible solution methods per problem are easily feasible with the //ELLPACK library, due to resource limitations we have generated data for only about $nmp_p \approx 5$), $nmc_c \approx 5$ (again, for the data we have been able to generate), $nr \approx 5$, $nc = 2$, $ne = 3$. That is, a grand total of about 8,000 rules were generated for the 275 problems.[4]

[4]While, this is what we would like to generate, we have yet only produced those rules necessary for the tests described earlier.

4.10 Conclusion

In this chapter we have described a computational intelligence paradigm to assist in the selection of a PDE solver that satisfies a user's objectives for error and computational cost. The selection is one of a multitude of (grid, method) pairs and it uses performance information from a population of PDE problems. This paradigm has been implemented in the form of an expert system for the //ELLPACK library. We have validated the accuracy of PYTHIA's predictions for two classes of PDE problems for which we know a posteriori rules for the performance of applicable solvers. We have also validated PYTHIA's class selection methodology for the neural network based scheme. The results indicate that PYTHIA's accuracy increases when the knowledge base (number of rules) increases. PYTHIA's selections almost coincide with the results of various known performance evaluation studies. PYTHIA's paradigm is seen to be a good alternative to support "intelligence" in PSEs for PDE based applications. PYTHIA is currently being extended to parallel elliptic solvers.

Chapter 5

SoftLab: Towards a Virtual Laboratory for Science

This chapter addresses the problem of building software environments that integrate physical *experimentation* (as performed in "wet" laboratories) and *computation* (as performed in "dry" laboratories) within one cohesive system called a *virtual laboratory*. The goal of virtual laboratories is to provide application scientists an environment in which they can apply experimentation and computation in a collaborative manner in their exploration of science. We describe a vision for how such environments can be used. Issues in designing and implementing virtual software laboratories are also considered. Finally, an example laboratory is described to illustrate the key concepts developed in this chapter.

5.1 Introduction

With recent advances in computer hardware and software, computational science is rapidly emerging as a third paradigm of science, augmenting the time–tested methodologies of theory and experimentation. With this revolution comes the need to integrate these diverse paradigms into cohesive systems that scientists can use to achieve their scientific ideals. While hardware/software for controlling laboratory instrumentation, gathering data and for analyzing that data generally exists today (see for example [Nat92] and [BH92]), there has been little work on integrating experimentation and computation in one cohesive environment. The general state–of–the–art in such integration is to manually (off–line) understand and apply the interactions between them (see for example [CWW91a] and [TK89]). Our own PDELab work concentrates on building problem solving environments for the *computational* aspects of a problem. In this work, we address the issues involved with building software environments that integrate experimentation and computation and describe a prototype implementation currently in progress.

The focus of SoftLab is to realize a virtual laboratory with a software layer above the familiar physical wet and dry laboratories of science and engineering. The goal

is to build the needed software and algorithmic infrastructure to realize some of the important activities in a lab (such as instrument configuration and experiment monitoring), and produce the electronic analog of a scientific laboratory environment for important applications. The real time requirements for the interactions between experimental or production processes on the one hand, and solving analytical models by computers on the other hand, necessitate addressing the fundamental challenge of harnessing the power of high–performance computing equipment and experimental instruments and exploiting them through very high level systems that are problem–oriented. Such systems require a wide range of expertise plus a flexible and diverse array of equipment. The SoftLab project combines the hardware facilities with software facilities (problem solving environments) to carry out scientific and engineering research. The facilities include high–performance graphics systems to support scientific visualization, geometric modeling and design, and multimedia graphical user interfaces for parallel programming and programming–in–the–large, as well as high–performance computing power to facilitate the excessive amount of computation needed to support the SoftLab goals.

The state of the art of integration of simulation and experimentation is somewhat primitive. For all but the simplest tasks, this type of integration currently occurs only via human intervention. Common manual approaches include plotting experimental data and then measuring from the plot to get quantitative data. In the reverse direction (i.e., from computation to experimentation), the simulation provides data on the expected behavior of the experimental process. By comparing this data with experimental data, it is possible to estimate parameters of the model that govern the situation and possibly use that to control the experiment.

We develop several usage scenarios for the virtual laboratory that integrates experimentation and computation as well as a methodology for mapping between these two paradigms. Then, we design and implement a virtual software environment that allow users to intelligently integrate the results of computation with experimentation and vice versa.

This chapter is organized as follows: in Section 5.2 we discuss others' work that is most relevant to us. Section 5.3 presents the vision behind the SoftLab project. Section 5.4 discusses the implementation aspects and Section 5.5 considers the software architecture of the virtual laboratory. Finally, Section 5.6 describes an example SoftLab that is currently being built.

5.2 Related Work

Work in several different areas is of relevance to the SoftLab project. For the physical laboratory aspects of SoftLab, we are interested in instrument control, data acquisition and data analysis. From the simulation aspects, we are interested in simulation and modeling laboratory experimentation.

Hardware and software for data acquisition and instrument control has seen much progress in recent years. Instrumentation interfacing got an early start with Hewlett–

Packard's 1965 HP–IB (HP Interface Bus) instrument interface bus specification which defined a standard hardware interface for laboratory instruments. In 1987, the HP–IB standard was generalized across vendors and became the IEEE 488.1 standard or the General Purpose Interface Bus (GPIB) standard. A later standard IEEE 488.2 strengthened this by precisely defining how controllers and instruments communicate. In 1990, Standard Commands for Programmable Instruments (SCPI) took the command structures defined in IEEE 488.2 and made a single, comprehensive programming command set that is used with any SCPI instrument. The newest addition to this family of instrument control equipment is VXI. Introduced in 1987, VXI is seeing rapid growth and acceptance worldwide. VXI uses a mainframe chasis with slots to hold modular instruments on plug–in boards. The VXI backplane includes a 32–bit VME bus as well as high–performance instrumentation buses for precision timing and synchronization between instrument components. [Nat]

Along with these hardware standards have come software that provide convenient access to instruments. The current state of the art in software for instrument control, data acquisition and data analysis is exemplified by LabView [Nat92]. The facilities available include graphical program development tools, graphical user interface building support, and a notion of "virtual instruments" (VIs). A VI is a software interface to the physical instrument; the software interface provides mechanisms to perform all the operations provided by the actual instrument. Various data analysis VIs are used to graphically analyze the data. The LabView package also includes drivers for controlling instruments via various access protocols, including GPIB and VXI.

Simulation of laboratory processes is one of the physical phenomena that is typically modeled with partial differential equations. Software for simulating laboratory processes belong to the class of PDE based application problem solving environments described in Chapter 2. In most current environments, as argued in that chapter, existing software is hand crafted for the specific model. Furthermore, the existing software foundation is generally a large FORTRAN program and not sophisticated software environments comprised of many well–defined parts.

5.3 The SoftLab Vision

The SoftLab environment has two *views* that it provides to its user: the experimental view (SoftLabView) and the computational view (SoftPDEView[1]). The experimental view gives users access to the SoftLab environment in terms of the experimental devices that they are familiar with in the wet laboratory. The computational view provides access in terms of the simulation environment. While at first these views appear to be completely disjoint, there is in fact significant interaction between them. These views must be active views; i.e., for example, while in the experimental view one must be able to actually perform an experiment, gather the data and visualize the

[1]While we use "PDE" here with the expected implication, it should be noted that the discussion in this chapter is independent of the specific type(s) of mathematical models that govern the experimental process(es) involved.

results: the goal is for the virtual laboratory to *automate* the existing physical one. Of course, some physical work is necessary to set up instruments and possibly even to control some apparatus. However, in the virtual laboratory of the future, it is not difficult to visualize intelligent robotic assistants performing the manual work in the wet laboratory and the scientist using purely the software–driven virtual laboratory.

5.3.1 Functionality

From the experimental view of SoftLab, the obvious functionality is to be able to (remotely) configure instruments, control the running experiment, gather data and analyze the data in various ways. In addition to controlling the running experiments and gathering data, one would also like to monitor the experiment either via real time visualization of the data and/or by direct visual means using video teleconferencing facilities. A second category of functionality evolves from the notion of *virtual instruments*[2]. After setting up an experiment in the usual way, it one *simulates* the instrument during the experiment, provided, of course, that appropriate (mathematical) models are available. The benefits are manifold; virtual instruments provide broad access to unavailable or expensive physical instruments. They are an excellent teaching/training tool to teach experimental methodology. Finally, it should be possible to "save" experiments for later use. An experiment is represented by the instrument configurations, control parameters and resulting data. Figure 5.1 shows an instance of the experimental view of an instance of a SoftLab.

From the computational view of SoftLab, the obvious requirement is to be able to simulate accurately the experimental and other related physical processes. This view need not be artificially separated from the laboratory view; wherever appropriate computational models are available, simulation instead of experimentation should be a choice available to the user. Computation also often adds to the results that are experimentally measurable. For example, in the Bioseparation example we describe later, the experimental devices can only measure the concentration at the outlet of a column. However, computation can also provide the distribution of the concentration within the entire column as well. Merging the results from the two therefore provides the SoftBioLab user with a better understanding of the process taking place within the column. Another advantage of simulation is that it is often possible to simulate more general, more complex, larger scale experiments than can be performed in a typical (academic) laboratory. This allows SoftLab users to experiment with scale issues in ways that were not feasible before. As with experimentation, it should always be possible to "save" a simulation for later use. Figure 5.2 shows an instance of the computational view of SoftLab.

[2]Note that our use of the term virtual instrument is different from LabView's use discussed earlier. Their view is of an instrument accessed via a software interface, ours is of a physically non–existant instrument.

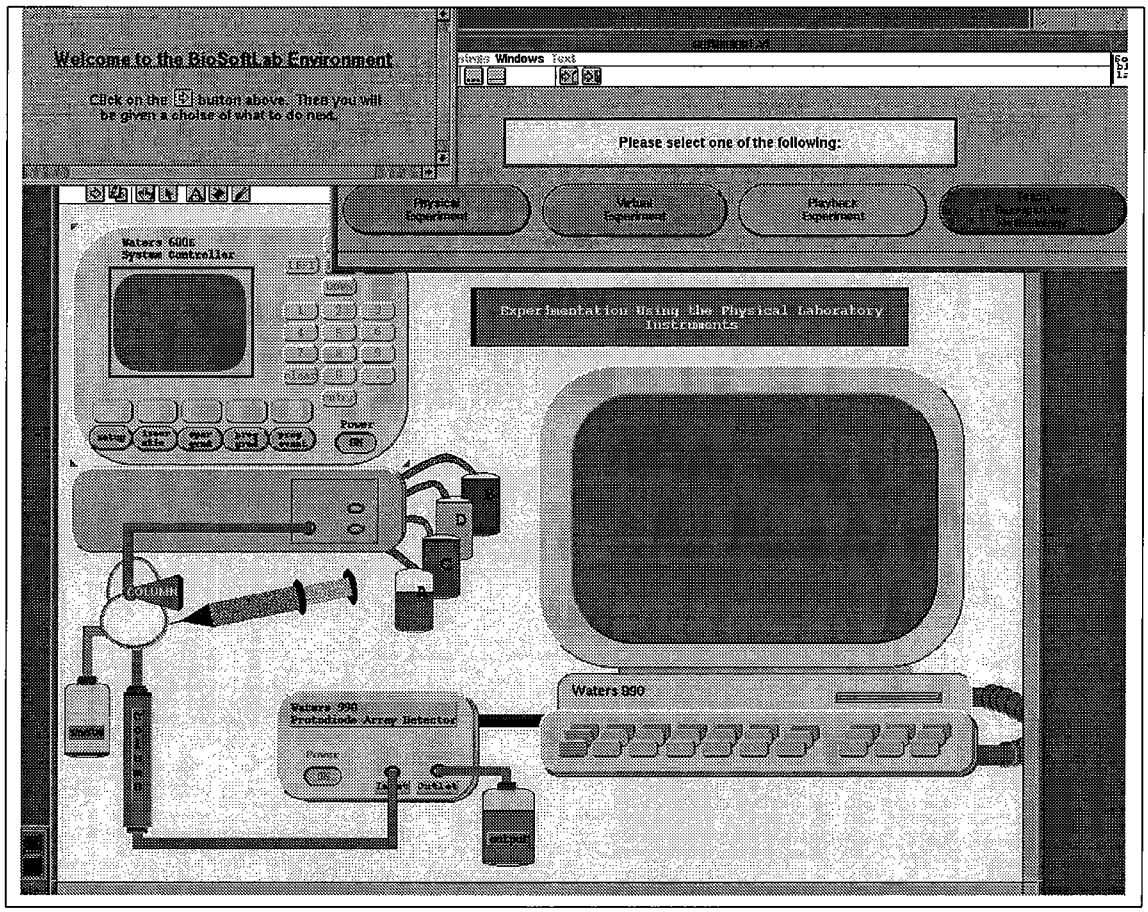

Figure 5.1: An example instance of the experimental view of SoftLab. Each component in the picture represents an actual physical instrument in a particular wet laboratory. The components are *active* components; clicking on the various buttons will perform the same action as the corresponding instrument and hence allows the scientist to use this interface to drive the experiment.

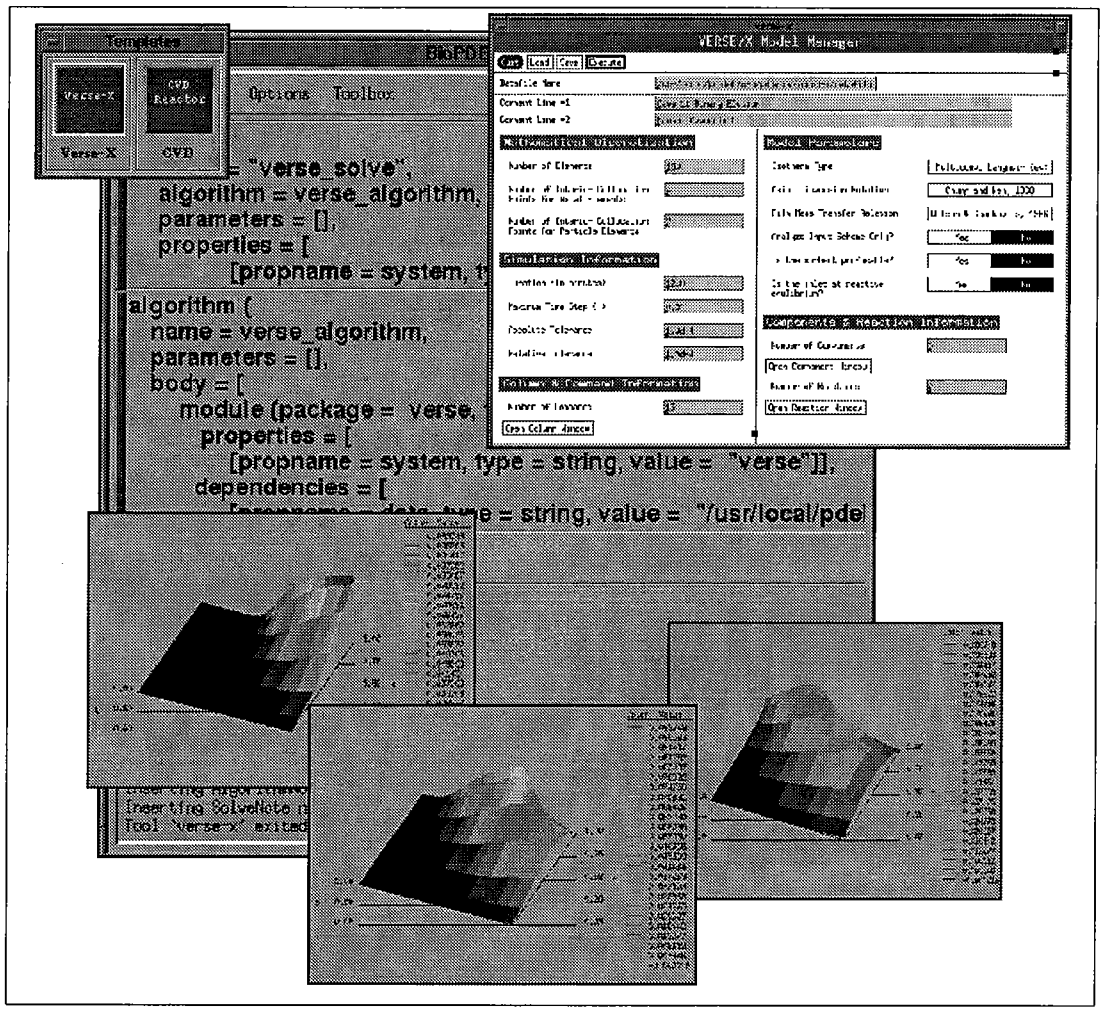

Figure 5.2: An instance of the computational view of SoftLab. The interface at the top right part of this figure is the application specific interface that allows application scientists to configure the simulation using terminology familiar to them. The other components shown in the figure are generic tools from PDELab that support the simulation and the visualization of the computed data.

5.3.2 Usage Scenarios

By combining functionality of the experimental and computational views of SoftLab, we identify five different usage scenarios. Each scenario corresponds to a realistic combination of information from these two views to provide the SoftLab user with an environment with much greater functionality than is possible today with the disjoint wet and dry laboratories. The existing SoftLab environment implements the first two and last scenarios. The virtual reality playback and teaching with multimedia presenation scenarios are currently under development.

Experimentation Using Physical Laboratory Instruments

In this scenario, the experimenter sits in front of the SoftLabView station which provides a view of the physical laboratory. Experimenters can run their experiments remotely using the SoftLabView interface to the physical laboratory. The remote functionality includes controlling experimental devices and instrumentation, monitoring the devices during the experiment, visualizing the operation of instrumentation and other experimental devices, and collecting the experimental data in real time.

All of these activities use the software representation of the actual laboratory devices, and they are operated in the *same* manner as the actual devices themselves. The visualization and monitoring of the experiment appear on the SoftLabView devices as they would in the physical lab.

The SoftLabView station supports one additional function that the physical lab cannot: recording the experiment for future use. The recorded experiment consists of information which covers the entire experimental process, including setup, monitoring, and data collection. This recorded experiment can be used to playback the actual experiment on the SoftLabView station. It can also be used as input into a transformation process linking experimental data with simulated data.

Experimentation Using Intelligent Simulation

In this scenario, the experimenter sits in front of the integrated SoftLabView + SoftPDEView station presenting the laboratory view. In addition, special interfaces for controlling the parameters of the computational model beyond the scope of the experimental devices are also provided. Since the simulated model generally allows far greater flexibility in the specification of the input than the experiment itself, there are numerous variables, definitions and specifications which belong solely to the simulation. These additional specifications allow the experimenter to scale–up, scale–down, optimize, mix or separate the input in ways that are not feasible in the physical laboratory.

However, since the parameters represent extrapolations of actual experimental input, wherever possible they are tied to the physical devices from which they were extrapolated. In this way, users of the simulation model will understand how each parameter of the model is associated with the actual experimental input.

When the model input has been specified, the simulation engine is started. The simulated experiment is monitored and visualized directly on the SoftLabView station as in the case of the physical experiment. But, in this case, the monitoring data is generated by the simulation instead of by the actual devices. This again allows the experimenter greater flexibility in visualizing the experiment, since the simulation generally produces intermediate data which the physical devices cannot. During the simulation, output data is generated and collected.

The entire simulated experiment can be recorded for future use. The recorded experiment consists of information which covers device setup, simulation input specifications, monitoring, and data collection. This recorded experiment can be used to playback the simulated experiment on the SoftLabView station. It can also be used as input into a transformation process linking simulated data with experimental data.

Virtual Reality Playback of Recorded Experiments

This scenario allows the experimenter to playback an experiment that was recorded earlier from within the SoftLab laboratory environment, during either a physical experiment or a simulated experiment. Again, the user sits in front of the integrated SoftLabView + SoftPDEView station. The user has two modes for accessing the experiment to playback.

The first mode allows the user to select the experiment from a database of saved experiments. Recorded experiments have identifiers and descriptions attached to them, and the user can select experiments from among these.

The second mode requires the experimenter to set up the experiment himself using the physical devices and, if appropriate, the simulation input interfaces. These setup and input specifications are then used to identify the experiment from the database of recorded experiments. If the input specifications can be matched to an existing experiment, the recorded experiment is loaded into the SoftLab laboratory and played back.

The playback facility allows users to run actual and simulated experiments in parallel, so that intermediate monitoring data and final results from both scenarios can be examined and compared. Furthermore, it provides a scenario for possible future education use; integrating virtual reality technologies, it may be feasible to teach experimental procedures using the SoftLab virtual laboratory rather than the physical one.

Teaching Experimental Methodology with Multimedia Presentations

The educational component of the SoftLab laboratory environment is a primary focus of the SoftLab research project. The use of the SoftLabView + SoftPDEView station for teaching experimental methodology is a unique and important addition to an engineering classroom.

The SoftLabView station which represents the instrumentation of the actual physical laboratory can be used by instructors to introduce students to the characteristics,

operation, and functionality of the physical devices. This can be done directly in the classroom, without entering the lab where the experiments actually take place. SoftLab can be used to teach the students how to set up lab instruments for various experiments, how to monitor the devices during operation, and how to collect results from the experiment.

Since SoftLab supports hybrid experimental and computational models, it can be used to run both models in parallel, link data from one model to the other, apply transformations on the experimental and computational data, and visualize the results. This process furthers an understanding of how the experimental process and the theoretical model that simulates it are related to each other.

The advantages having of this capability in the classroom are manifold. It increases expertise in laboratory instrument operation on a large scale and without use of the costly lab. It allows access to distant laboratories. It allows beginners to perform "dangerous" experiments. It uses a combination of recorded actual and simulation data to teach how these two modules interact. It uses simulation to visualize and explain phenomena not measurable by real–time experimentation. These and many other benefits result from a successful application of SoftLab to a specific application in the engineering sciences.

Simulation of the Experiment's Mathematical Models

PDELab can also be used build custom environments for particular applications or models. The customized environment, SoftPDEView, can be tailored specifically to the computational models that represent a particular enginnering application.

For a given application, the SoftPDEView station presents a specialized user interface to the simulators used to model the experimental process. It communicates in terminology that is familiar to the experimenter and supports problem specification with all appropriate editors and tools from the large collection available from the general PDELab PSE development framework.

Software for the solution of the model may or may not exist in the original engineering laboratory. If there is a solution process in place for the particular theoretical model, it can be integrated into the customized SoftPDEView environment. It is also possible to solve the model equations using other techniques available from the PDELab libraries of solvers, including parallel methods. All appropriate methods, or any subset thereof, can be integrated into the custom environment for a particular engineering application.

5.4 Implementation of SoftLab

In this section we consider some of the issues that must be dealt with in order to implement the SoftLab vision described earlier. In the two broad views mentioned earlier (laboratory view and computational view), there are four tasks that must be supported in order to build the SoftLab environment: gathering experimental data

and controlling experiments, simulating experimental processes, integrating experimentation with simulation, and saving experiments and computational results.

5.4.1 Gathering Experimental Data and Controlling Experiments

The methodology for interacting with instruments and the types of interaction possible depends on the access technology present in the instrument. For some legacy equipment, no remote interaction is possible. For others, the only interaction is via serial lines (RS–232 ports). Newer equipment have GPIB or VXIbus interfaces for complete remote access. Since the SoftLab station (consisting of the SoftPDEView and the SoftLabView stations) is expected to be a relatively high–end desktop workstation, instrument access is preferably provided over normal data networking facilities. In fact, ethernet (and other network) interfaces for GPIB devices as well as corresponding cards for the workstations are currently available. Since data gathering is a real time process, it is sometimes necessary to use dedicated hardware and/or special software (such as real–time versions of operating systems) in order to meet the stringent timing requirements. In any case, the resulting data is made available in some representation for real–time visualization and analysis or later processing. Figure 5.3 illustrates three possible hardware configurations for data acquisition and instrument control.

Once the hardware facilities have been identified, the software must be considered. The available software infrastructure for device control include the necessary device drivers for various types of machines and operating systems. Starting with the drivers, one can implement the protocol(s) supported by the instrument to provide access to the instrument. However, existing systems such as LabView provide many tools that allow developers to start at a level much higher than raw device drivers. LabView provides implementations of various industry standard protocols such as the GPIB protocol as well as tools for implementing special protocols over low–level mechanisms such as RS–232. Using these tools, it is relatively easy task to build drivers that communicate with instruments adhering to the timing requirements of the instruments. For (legacy) instruments that can only communicate with special purpose hardware, it is necessary to work around the software provided by the vendor and access the data after processing by the custom software.

5.4.2 Simulating Experimental Processes

The SoftLab environment expects to be able to simulate the laboratory procedures using mathematical models of the processes involved. Application scientists must (manually) develop the mathematical models of the physical processes. Often, partial differential equations are a natural model for representing physical processes and hence PDELab becomes a useful tool during this process.

Once the models have been developed, custom solvers and application specific

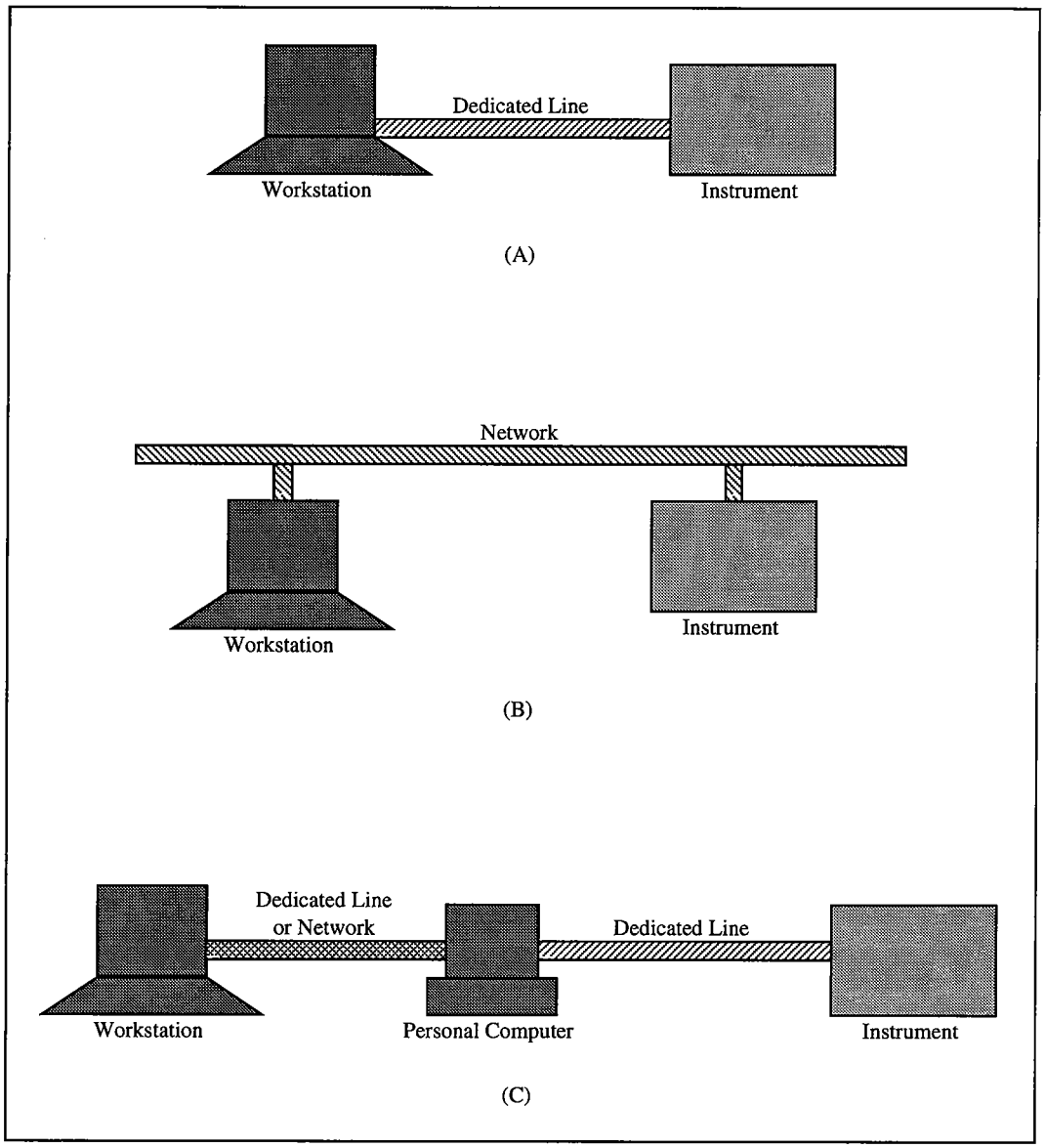

Figure 5.3: Three possible hardware configurations for data acquisition and instrument control. In configuration (A), a workstation is connected via a dedicated line to the instrument using dedicated hardware and software. In configuration (B), the workstation is connected to the instrument via some network. Special hardware is needed at the instrument to interface directly to the network and the workstation requires special software. In configuration (C), a (presumably smaller) personal computer is used as a dedicated host charged with the task of interacting with the instrument. The personal computer could then be connected to the workstation via a dedicated line or via some network.

problem solving environments must be developed using PDELab. Since the physical processes are often transient, nonlinear processes, it is imperative that high performance computing techniques be used to solve the models to achieve the high accuracy solutions needed quickly for the SoftLab station. Thus, the application PSE developed using PDELab forms the backbone of the SoftPDEView station.

5.4.3 Integrating Experimentation with Simulation

To integrate experimentation and simulation, the application scientists must abstract and formalize the interactions. Then, simulation parameters must be mapped to experiment parameters and vice versa. Often most of the interactions are heuristic and judgemental in nature. Then, it is necessary to use some heuristic reasoning technique such as rule based systems or neural networks in order to automate this interaction.

5.4.4 Saving Experimental and Computational Results

For playback and educational use, SoftLab requires that experiments and simulations be saved for later access. To achieve this, one must record instrument configurations, experimental input parameters, simulation parameters, and, of course, experimental and simulation results in a database. Various alternate search and access schemes must be available to facilitate all the later reuse needs of SoftLab.

5.5 Software Architecture

The software architecture of the SoftLab environment that is currently being developed is described in this section. Figure 5.4 illustrates the structure of the SoftLab environment at a high–granularity level. The PDELab software infrastructure for building problem solving environments is the starting point for SoftLab. There are two main components in the current prototype implementation; LabView and PDELab. LabView provides the mechanisms for interacting with instruments and PDELab provides the mechanisms for solving the PDE models as well as for building the application problem solving environment that integrates all the components.

The SoftLab architecture is a five–layered architecture that adds two more levels to the PDELab model and broadens the lower levels as well. Figure 5.5 illustrates this architecture.

5.6 SoftBioLab: The Bioseparation SoftLab

Chromatography is the process of separating components in a fluid by passing a solution mixture through an absorbent column so that each component adsorbs to the surface differently than the others, and thus eludes at different times [CWW91b]. The

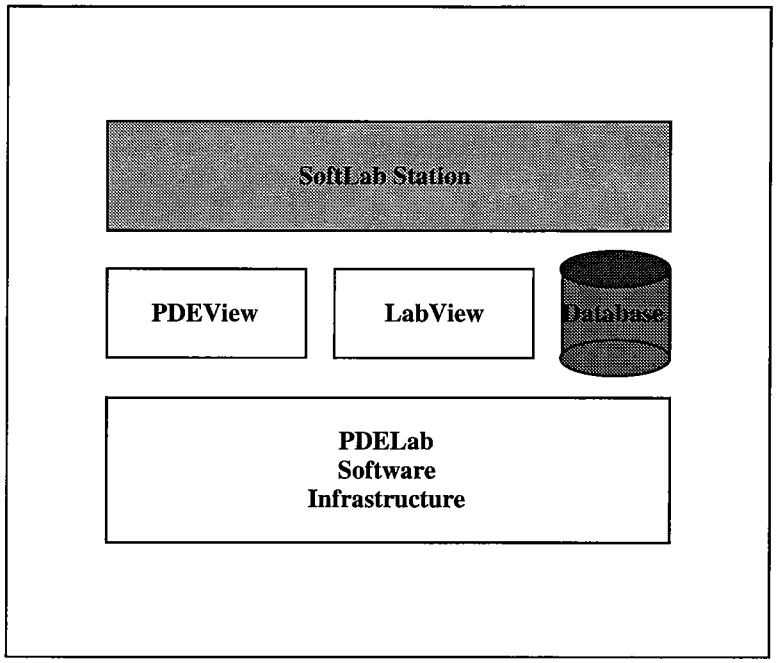

Figure 5.4: Structure of the SoftLab environment. LabView itself does not depend on the PDELab software infrastructure directly, but the interconnection of LabView to the rest of the environment is built using tools provided by PDELab. The SoftLab station consists of the SoftLabView station and the SoftPDEView station.

process is used for a final purification of proteins, chemicals and biochemicals used for the manufacturing of pharmaceutical and food products, water treatment and many other biochemical processes. Chromatography of biological molecules is often quite complex, and many of the dynamic aspects are not well understood. Preparative chromatography uses high concentration with many solutes in the feed which introduces interference (solute competition for sorbent sites) and nonlinear isotherm effects. Some components involved in a system may react with themselves (homo–aggregation) or each other (hetero–aggregation), which may result in the generation of new species. Also, effects such as pH changes are often used to aid in the separation of biochemicals. Some solutes also undergo reactions on the solid phase, such as denaturation, or have rates of adsorption and desorption that are significantly slower than mass transfer rates, such that equilibrium isotherms are no longer valid. Any or all of these complications may be present in a given system, and each must be fully understood in order to model adequately the complete process.

To optimize output, it is generally beneficial to know precisely when and at what concentration a particular component can be found. For this purpose, the process must be modeled and accurately described. With so many components and time-dependent processes occurring in chromatographic and adsorption processes, the resulting effluent history is usually difficult to predict when feed concentrations fluctuate, or when operating concentrations are changed for scale–up or optimization.

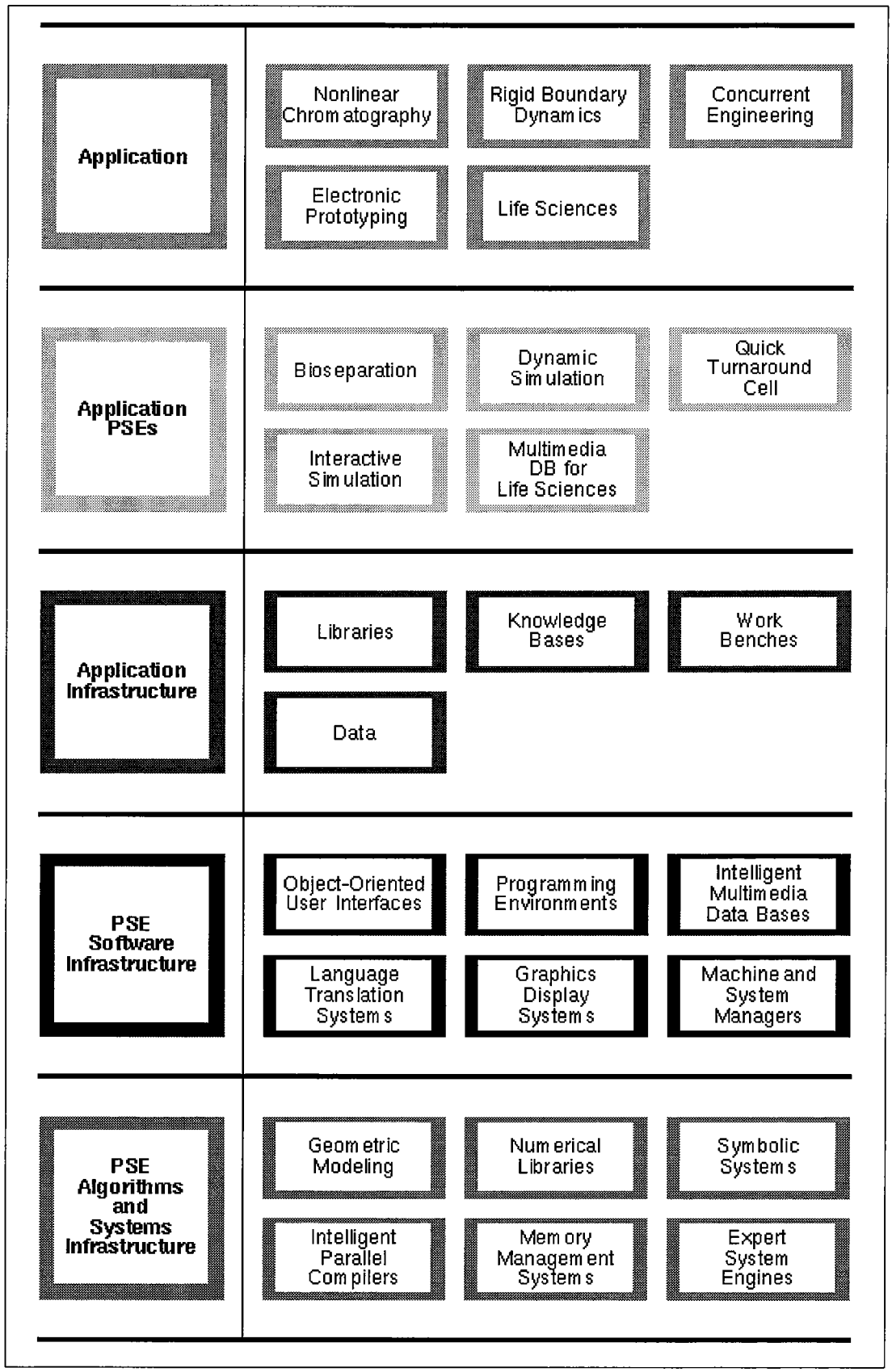

Figure 5.5: Software architecture of SoftLab.

Furthermore, when biological solutes are involved, experimental studies become expensive and complex. In light of those challenges, computer simulation of these complex systems becomes very attractive. Experiments are still required, but feedback from computer modeling can greatly reduce the number of experiments needed to understand and optimize the process.

In [BWZW91], Berninger, Whitley, Zhang and Wang develop a versatile partial differential equation model for simulating reaction and nonequilibrium dynamics in multicomponent fixed–bed adsorption processes. Using this model and the corresponding implementation as a basis, we are developing an instance of the SoftLab environment for bioseparation processes. The existing physical laboratory is brought into the SoftLab model and then the corresponding simulation is integrated with this environment. The rest of this section describes first the physical laboratory and then the virtual laboratory we are developing.

5.6.1 The Physical Laboratory

The bioseparation laboratory operated by Professor Linda Wang is situated in the School of Chemical Engineering. It currently consists of three isolated personal computer (PC) monitored experimental facilities and a workstation. See Figure 5.6 for a block diagram view of the equipment in the bioseparation laboratory.

The input solution mixtures are placed in reservoirs connected to the pump controller, a Waters 600E that allows the experimentor to control the mixing of the various solutes and their flow rates. The mixed solution is sent through a column where separation takes place. The output of the column is sent to the detector, a photo diode array which produces a number of analog values representing the concentrations of various components as they elude from the column. These values are then sent to a strip chart recorder for continuous output and to a personal computer (an NEC 80286) where it is received by special hardware and processed by custom software. The custom hardware in the PC is basically an analog–to–digital convertor card and the software provides convenient access to the data, including in the form of a file of numbers.

On the simulation side, the bioseparation group has a custom implementation of a numerical simulator called VERSE [BWZW91] that solves the experiment's model (partial differential) equations. Simulations are run on a Sun workstation and take on the order of hours to execute. In the current environment the interaction between the experimental instruments and the numerical models is done manually.

5.6.2 Description of the SoftBioLab

The SoftBioLab system integrates the various components of the physical bioseparation laboratory to provide users with a virtual laboratory that implements the SoftLab vision. Of the five scenarios developed present in the SoftLab vision, only three have yet been implemented: experimentation using physical laboratory instruments,

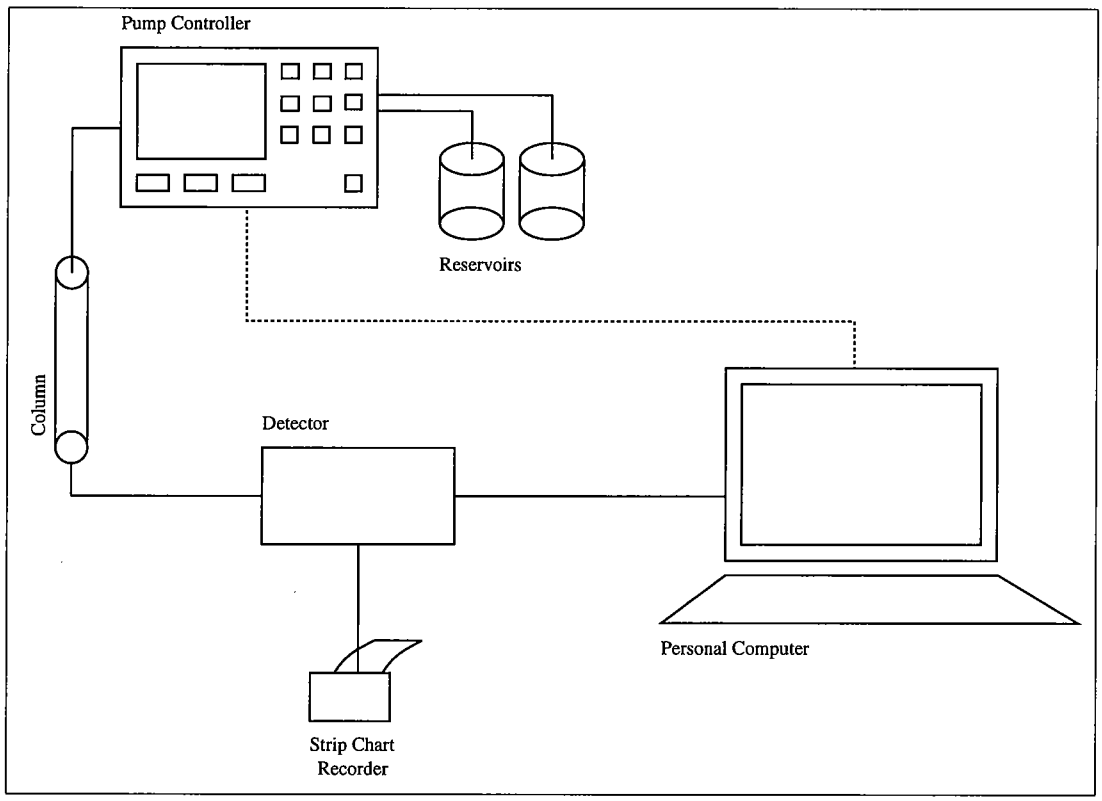

Figure 5.6: A block diagram of the equipment in the Bioseparation Laboratory in the School of Chemical Engineering. The pump controller regulates input from the reservoirs into the column according to how the controller is configured. As the components separate and flow out from the column, the detector measures the eluding concentrations and generates a trace on a strip chart recorder and also sends the information via a special card on the personal computer to the software running on that machine.

experimentation using intelligent simulation and simulation of the experiment's mathematical models. We describe each case below.

Experimentation Using Physical Laboratory Instruments

This scenario consists of the components for setting up and controlling the instruments, and for monitoring the instruments and gathering interesting data. The laboratory hardware is first made accessible to the SoftBioLab system.

The hardware configuration of the physical laboratory is shown in Figure 5.6. Since the instruments used in the bioseparation laboratory are somewhat dated, they do not support the GPIB or VXIbus standards. Hence, they needed to interact with the custom hardware and software installed on the personal computer. Thus, the third configuration of Figure 5.3 was used to interface with the hardware: the PC was placed on an ethernet network for remote access while the instruments were left in the existing configuration. Then, a LabView VI for developed for each instrument. These VI's present to the user the same interface as the real instrument and allows the user to perform experiments in the virtual environment in exactly the same manner as in the real environment. To implement this, SoftBioLab transfers the configurations of the physical instruments via the PC. Interactions with the PC occur via the ethernet network and the serial line connections between the PC and the SoftLab station. Figure 5.7 shows a picture of the physical pump control as well as the pump controller virtual instrument. Figure 5.8 shows the configuration of the virtual bioseparation laboratory.

A SoftBioLab user starts the experiment procedure by configuring the pump controller and other instruments appropriately. Then, the experimental compounds must be (physically) loaded into the reservoirs etc., that are located in the physical bioseparation lab. When the user activates the experiment, the instruments are (remotely) instructed to start and data is gathered directly onto the PC. With this SoftLab, real time monitoring is not possible as the raw data gathered from the instruments must first be processed by the (custom) software on the PC. The experiment is monitored remotely using video teleconferencing facilities which are also a part of this SoftLab. After the experiment is complete, the PC software generates the measured data in the form of a file of tuples of numbers. This file is then transferred over to the SoftLab station for visualization, analysis and recording.

The SoftBioLab station allows the user to animate the recorded data or to visualize it directly. If computed data is also available, then that data can be viewed simultaneously. Finally, all of the experiment information, including the instrument configurations, data gathering parameters and the data itself, can be saved for later use.

Experimentation Using Intelligent Simulation

This scenario uses the same user interface as above except that the physical laboratory is not used. After setting up the instruments as in the previous case, the user must

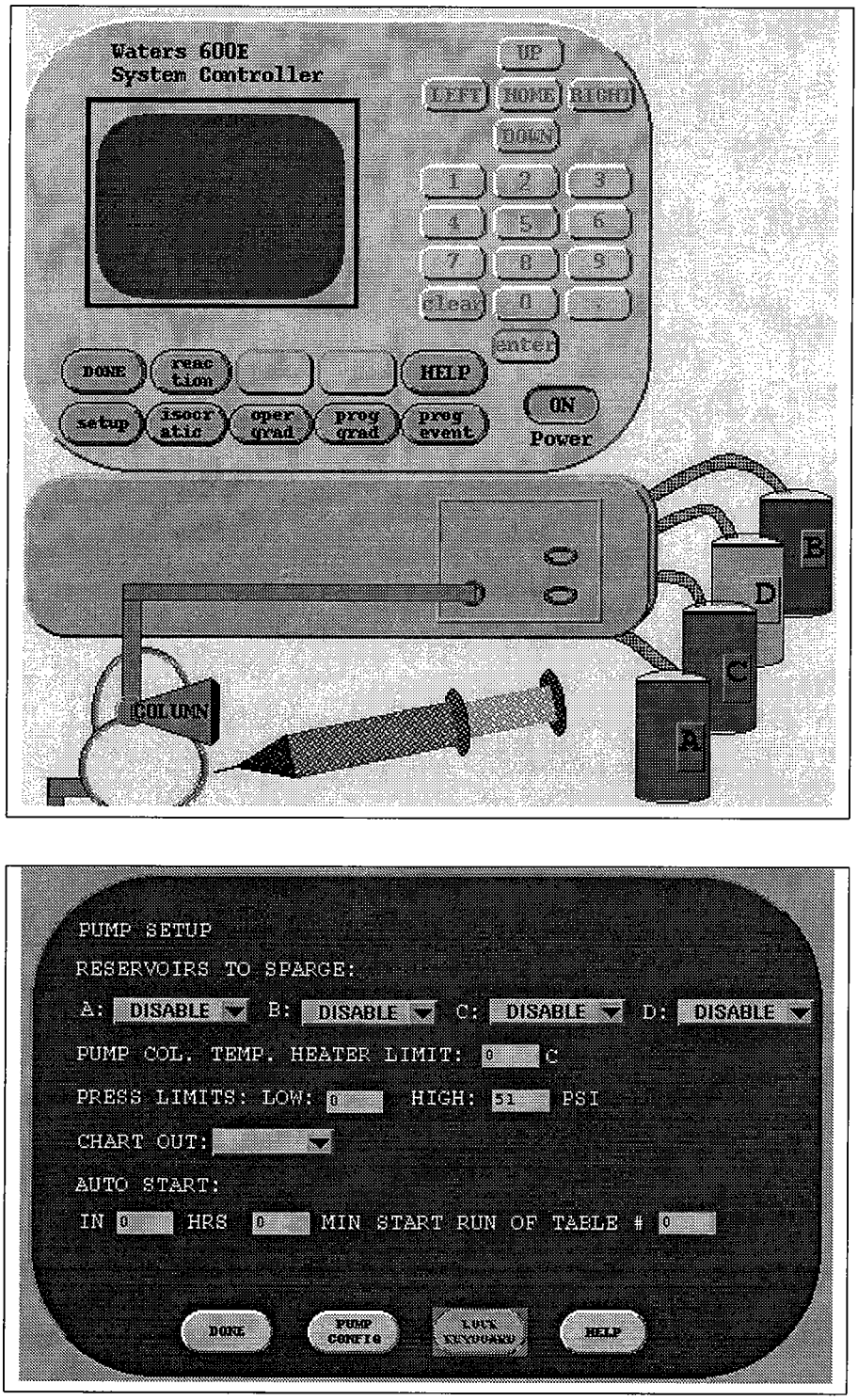

Figure 5.7: An instance of the pump controller virtual instrument. The top component is a view of the controller itself and the bottom illustrates the process of configuring the pump controller.

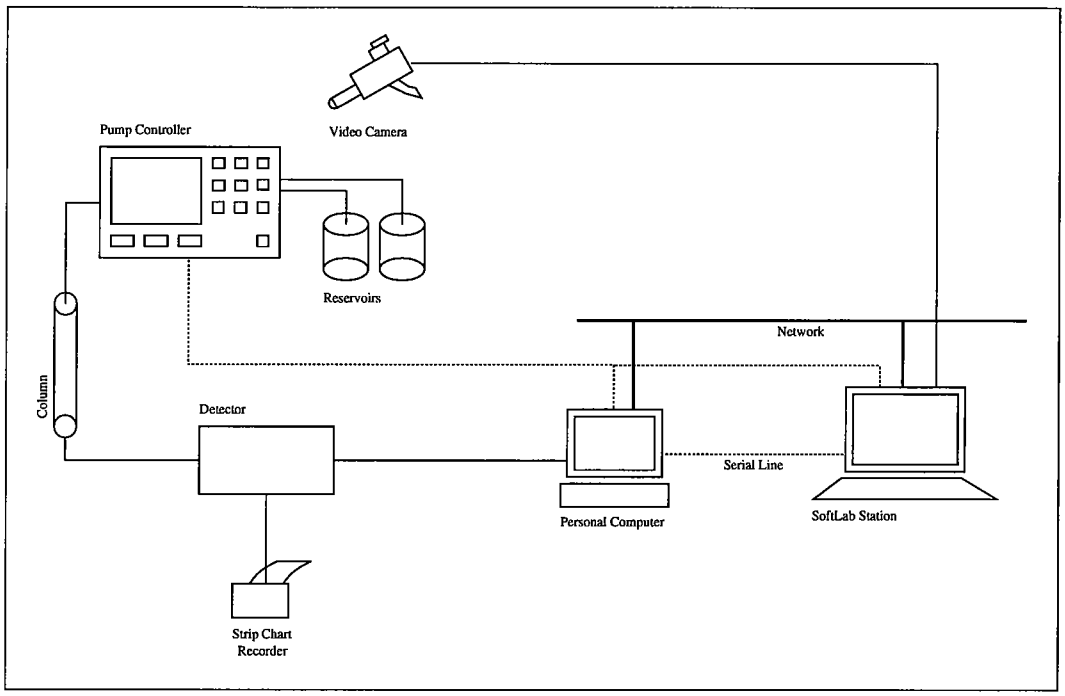

Figure 5.8: The configuration of the virtual bioseparation laboratory.

also set up some additional computational parameters that are needed to successfully simulate the experiment the user is running. Finally, when the user requests that the experiment be performed, the simulator is invoked with the appropriate configuration. Figure 5.9 shows an instance of the simulator VI which is used to specify the additional information.

While the computation is proceeding, the user may monitor the simulation using laboratory view. That is, the data generated by the computation is translated by the visualization VI to corresponding experimental data and then displayed as in the previous case. Figure 5.10 shows an instance of visualizing computed and experimentally gathered data simultaneously. Once the simulation is complete, the user can save all the relevant information for later use.

Simulation of the Experiment's Mathematical Models

This scenario uses the computational environment to allow SoftBioLab users to experiment using the mathematical model of the procedures performed in the physical laboratory. The interface is based on the PDELab environment and allows uses to configure the simulation in ways that may not be possible in the physical laboratory. Figure 5.11 shows an instance of this view. The data generated by the simulator can be saved for later analysis within this scenario or for use in the other scenarios.

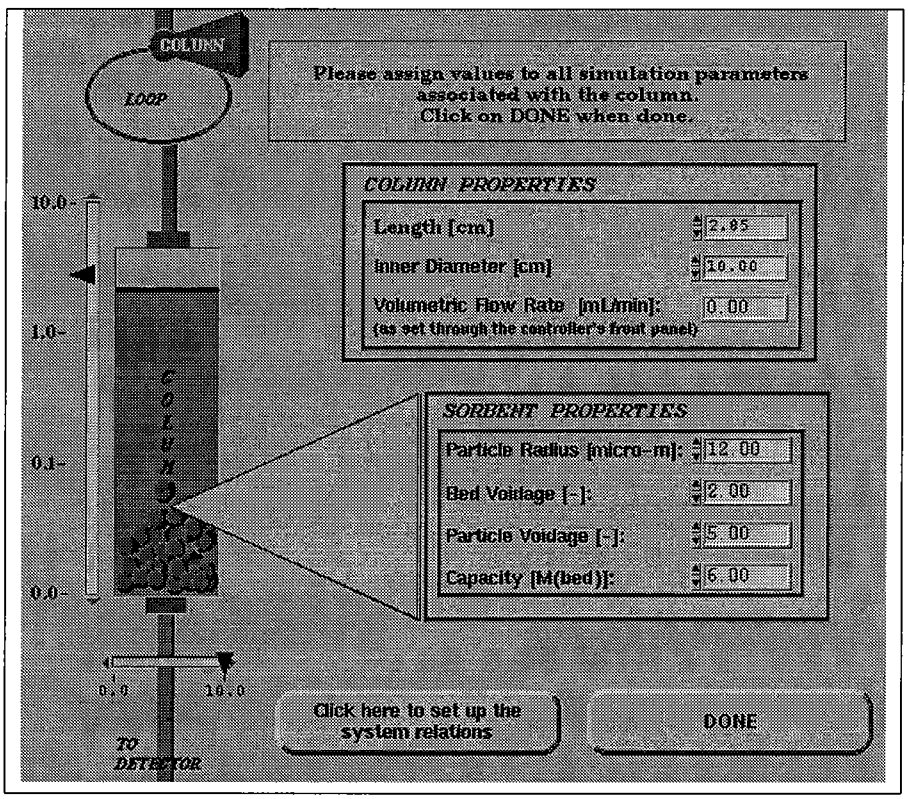

Figure 5.9: An instance of the simulator virtual instrument. In this case, the user is setting up the column properties and properties of the sorbent.

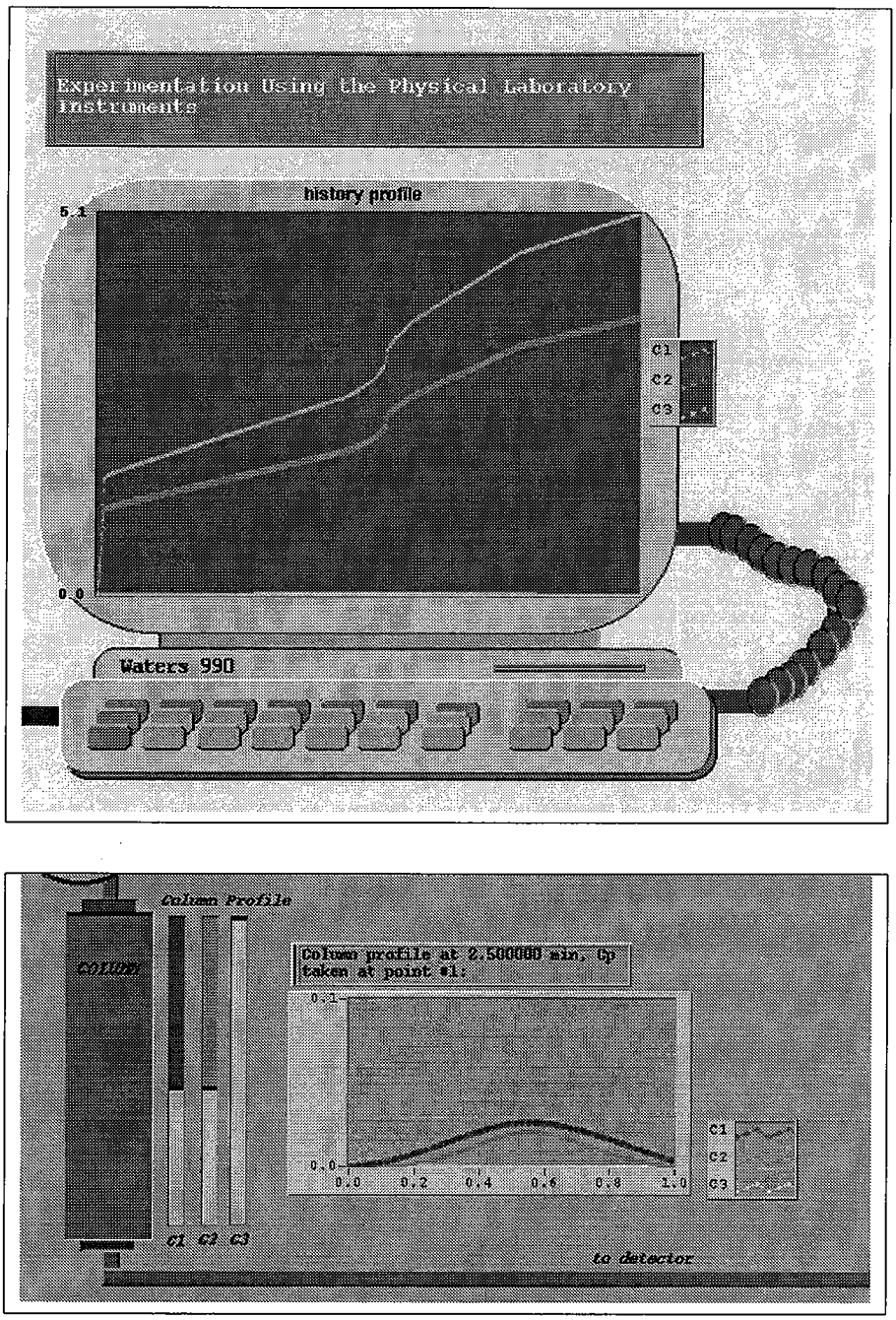

Figure 5.10: An instance of simultaneously visualizing computed and experimentally gathered data. The top component shows a visualization of experimentally gathered effluent history while the bottom shows a visualization of computationally obtained column profiles. A column profile shows the concentration of various components along the length of the column, which cannot be measured experimentally.

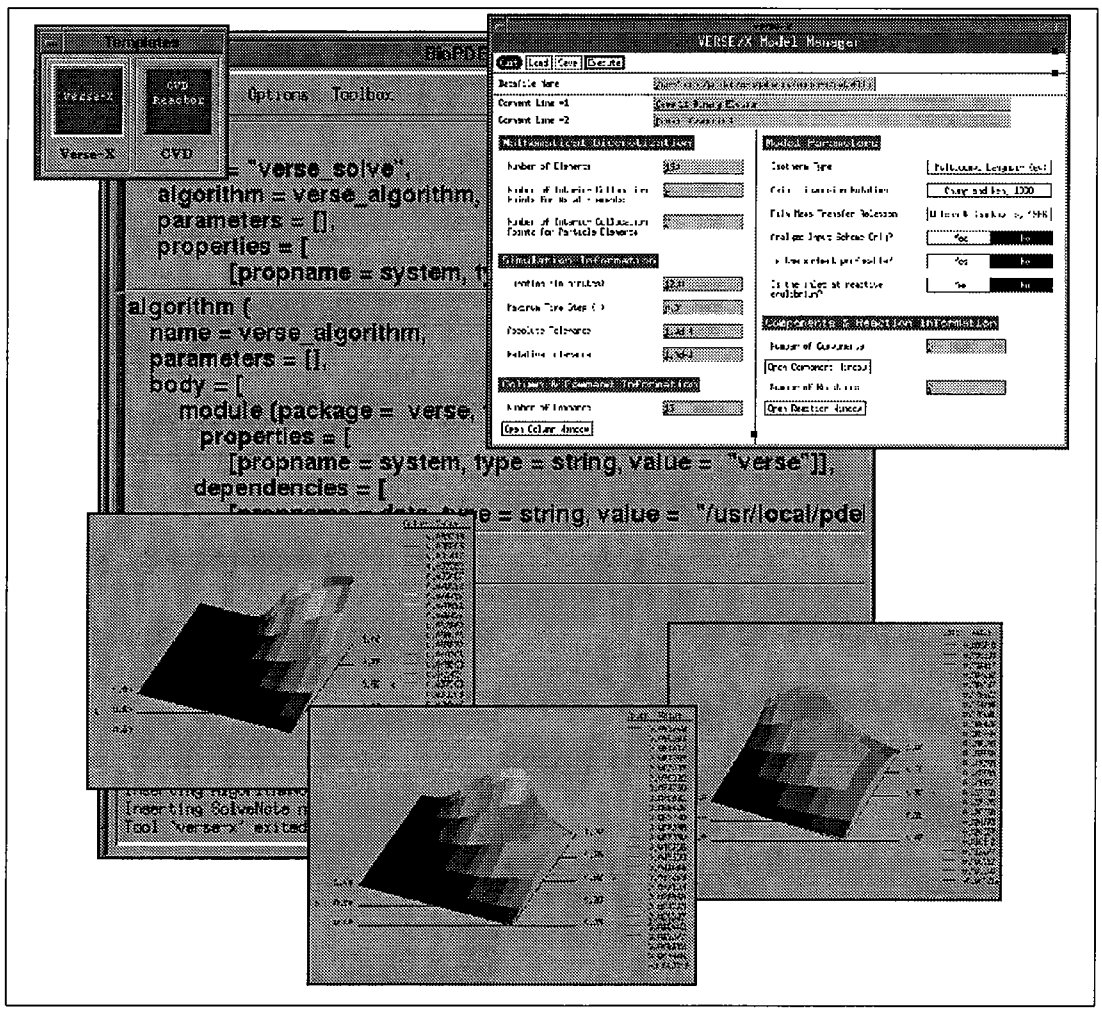

Figure 5.11: An instance of the computational view of SoftBioLab. The interface at the top right part of this figure is the application specific interface. The worksheet editor and the visualization tools are standard tools available in PDELab.

5.7 Conclusion

In this chapter we moved up a level from dealing with the internals of problem solving environments and address the task of building PSEs that integrate physical experimentation and computation. These integrated environments are implemented as a virtual software environment that provides a virtual layer over the familiar wet and dry laboratories of science. The environment is based on developing software interfaces with laboratory equipment and on developing mathematical models of the experimental processes. These components are integrated using the problem solving environment development framework originally developed for PDELab and augmented for SoftLab. An example SoftLab built for the Bioseparation Laboratory in the School of Chemical Engineering is also described.

Chapter 6

Conclusion and Future Work

6.1 Overview of the Thesis

This thesis has considered the problem of building problem solving environments (PSEs) for partial differential equation based applications. We have investigated four aspects of this problem: software architecture of development frameworks, high level languages, computational intelligence and integrating experimentation and computation. The introductory chapter laid the groundwork for the dissertation by briefly introducing the reader to partial differential equations (PDEs) and PDE based models, to problems solving environments and of course to the prototype system that is currently being built.

In Chapter 2 we presented the software framework of PDELab for building problem solving environments for applications based on PDE models. As a framework, PDELab includes significant PDE solving functionality as well as many tools and a methodology for building application PSEs. The PDELab environment is validated by several application PSEs that are being built.

In Chapter 3 we developed a high level symbolic language PDESpec for specifying PDE problems and their solution schemes. The language is based on decomposing the PDE problem into its constituent parts and on decomposing the solution process into a sequence of transformations. With the integrated symbolic–numeric processing environment, PDESpec becomes a powerful mechanism to compose new solution schemes from existing software components. The syntax of the language is based on the syntax of MACSYMA, a computer algebra system. We partially define the semantics of PDESpec and also discuss the design and implementation of a compiler and a run–time system for it. Several examples are provided to validate the power of PDESpec as a specification language.

In Chapter 4 we presented PYTHIA, a system for applying computational intelligence in problem solving environments. The algorithm selection problem is the problem of selecting an algorithm from an available set of choices so that the application of that algorithm would solve some problem within some *a priori* specified constraints. The PYTHIA system uses exemplar reasoning based on the prior performance of so-

lution methods on "similar" problems as a basis for predicting the performance of various methods. The "best" method itself is selected by some approximate match of the characteristics of the problem at hand with those of previously solved problems. The methodology used in PYTHIA is validated by several tests using a population of elliptic PDE problems.

In Chapter 5 we present the SoftLab framework for building problem solving environments that integrate physical experimentation as performed in "wet" laboratories and computation. The goal is to develop virtual laboratories that allow scientists to apply computation as a third paradigm of scientific investigation in an integrated manner. We describe a vision for how such environments can be used and then consider the design and implementation aspects of such software environments.

6.2 Future Work

A key aspect of the work on PDELab is the software architecture of problem solving environments. Due to the diverse nature of the components involved in an application PSE, one needs a very flexible, extensible architecture that supports the integration of mixed–language software components and access to diverse hardware ranging from workstations to parallel supercomputers to laboratory equipment. We plan to continue our research into software architectures with emphasis on PSE development frameworks. The development of *de–facto* standards for representations of PDE objects is another aspect of PDELab that we expect to continue to address in the future.

The PDESpec language work is a springboard for research into integrating symbolic and numeric computation. Our plans in this work include the development of pragmatic solutions to well understood problems in this area (for example, the symbolic generation of discretization error estimators and stencil–based symbolic operator discretizations) as well as the development of general methodologies and languages for efficient integration of these two computing paradigms. The further development of PDE solution transformations to a lower granularity to include so–called "point transforms" is also planned.

The pragmatic integration of computational intelligence in problem solving environments is crucial to the success of PSEs. We plan on further developing the PYTHIA system as well as on developing other reasoning systems for dealing with similar problems such as the linear solver selection problem. The ability to verify the compatibility of a sequence of transformations used in the solution of a problem (as in PDESpec) is essential to the success of the transformation composition approach to PDE solving. We plan to develop integrating facilities for automatically detecting mismatches and for suggesting alternates (by interacting with systems such as PYTHIA).

The SoftLab work opens up many possibilities for computational science. By integrating computation with the time tested methodologies of experimentation and theory, scientists are accorded views of their explorations than are often physically impossible. The educational possibilities of the SoftLab vision are also interesting;

students could potentially use virtual reality to understand physical processes without ever having to step into a "wet" laboratory. We plan on continuing our work into the problem of building software that realizes these and other possibilities. The key areas to be addressed include the software architecture of such environments (which is a generalization of the PDELab work), developing strategies for merging experimentation with computation and developing virtual reality based user interfaces for interacting with the virtual laboratory of the future.

Bibliography

[AB94] V. Anupam and C. Bajaj. Shastra: An architecture for development of collaborative applications. *International Journal of Intelligent and Cooperative Information Systems*, 1994. to appear.

[AM90] M. Ader and S. McMahon. The ITHACA technology: A landscape for object-oriented application development. In *ESPRIT '90 Conference Proceedings*, pages 31–51. Kluwer Academic Publishers, 1990.

[Bar86] Ray Bareiss. *Exemplar–Based Knowledge Analysis.* Academic Press, 1986.

[BBP+92] P. Baras, J. Blum, J. C. Paumier, P. Witomski, and F. Rechenmann. EVE: An object–centered knowledge–based PDE solver. In E. N. Houstis, J. R. Rice, and R. Vichnevetsky, editors, *Expert Systems for Scientific Computing*, pages 1–18. North–Holland, 1992.

[BD90] John Bonomo and Wayne Dyksen. XELLPACK: An interactive problem solving environment for elliptic partial differential equations. In E. N. Houstis, J. R. Rice, and R. Vichnevetsky, editors, *Intelligent Mathematical Software Systems*, pages 331–342. North–Holland, 1990.

[BDG+92] Adam Beguelin, Jack Dongarra, Al Geist, Robert Manchek, and Vaidy Sunderam. A users' guide to PVM: Parallel virtual machine. Technical Report ORNL/TM–11826, Engineering Physics and Mathematics Division, Mathematical Sciences Section, Oak Ridge National Laboratory, 1992.

[BG92] C. Bischoff and A. Griewank. ADIFOR: A FORTRAN system for portable automatic differentiation. Technical Report MCS–P317–0792, Mathematics and Computer Science Division, Argonne National Laboratory, 1992.

[BH92] Douglas C. Beethe and William L. Hunt. A visual engineering environment for test software development. *Hewlett–Packard Journal*, pages 72–77, 1992.

[Bir91] Kenneth P. Birman. The process group approach to reliable distributed computing. Technical Report TR91–1216, Department of Computer Science, Cornell University, 1991.

[Bis88] R. Bisiani. A software and hardware environment for developing AI applications on parallel processors. In Alan H. Bond and Les Gasser, editors, *Readings in Distributed Artificial Intelligence*, pages 451–456. Morgan Kaufmann Publishers, Inc., 1988.

[BL93] T. Berners-Lee. Uniform Resource Locators: A unifying syntax for the expression of names and addresses of objects on the network. (Draft) Internet RFC from http://info.cern.ch at hypertext/WWW/Addressing/URL/Overview.html, 1993.

[BRH79] Ronald F. Boisvert, John R. Rice, and Elias N. Houstis. A system for performance evaluation of partial differential equations software. *IEEE Transactions on Software Engineering*, SE–5(4):418–425, 1979.

[BWZW91] J. A. Berninger, R. D. Whitley, X. Zhang, and N.-H. L. Wang. A Versatile model for simulation of reaction and nonequlibrium dynamics in multicomponent fixed–bed adsorption processes. *Computers in Chemical Engineering*, 15(11):749–768, 1991.

[CF63] G. J. Culler and B. D. Fried. An on–line computing center for scientific problems. In *Proceedings of the IEEE Pacific Computer Conference*, page 221, 1963.

[CK70] A. F. Cardenas and W. J. Karplus. PDEL - a language for partial differential equations. *Communications of the ACM*, 13(3):184–191, March 1970.

[CW88] Thomas Cooper and Nancy Wogrin. *Rule–based programming with OPS5.* Morgan Kaufmann, 1988.

[CWW91a] K. E. Van Cott, R. D. Whitley, and N.-H. L. Wang. Effects of temperature and flow rate on frontal and elution chromatography of aggregating systems. *Separt. Technol.*, 1:142–152, 1991.

[CWW91b] K. E. Van Cott, R. D. Whitley, and N.-H. L. Wang. Effects of temperature and flow rate on frontal and elution chromatography of aggregating systems. *Separat. Technol.*, 1:142–152, 1991.

[DG89] Wayne R. Dyksen and Carl R. Gritter. Elliptic expert: An expert system for elliptic partial differential equations. *Mathematics and Computers in Simulation*, 31:333–343, 1989.

[DG92] Wayne R. Dyksen and Carl R. Gritter. Scientific computing and the algorithm selection problem. In E. N. Houstis, J. R. Rice, and R. Vichnevetsky, editors, *Expert Systems for Scientific Computing*, pages 19–31. North–Holland, 1992.

[DHHW93] Jack. J. Dongarra, Rolf Hempel, Anthony J. G. Hey, and David W. Walker. A proposal for a user–level message passing interface in a distributed memory environment. Technical Report ORNL/TM–12231, Oak Ridge National Laboratory, 1993.

[DRR88] Wayne R. Dyksen, Calvin J. Ribbens, and John R. Rice. The performance of numerical methods for elliptic problems with mixed boundary conditions. *Numerical Methods for Partial Differential Equations*, 4:347–361, 1988.

[ELHR88] L. D. Erman, J. S. Lark, and F. Hayes-Roth. Abe: An environment for engineering intelligent systems. *IEEE Transactions on Software Engineering*, 14:1758–1770, 1988.

[Fou92] Open Software Foundation. Distributed computing environment: An overview. Technical report, OSF, January 1992.

[GHPW92] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. A users' guide to PICL: A portable instrumented communication library. Technical Report ORNL/TM–11616, Engineering Physics and Mathematics Division, Mathematical Sciences Section, Oak Ridge National Laboratory, 1992.

[GHR92] Stratis Gallopoulos, Elias Houstis, and John Rice. Future research directions in problem solving environments for computational science. Technical Report CSD–TR–92–032, Department of Computer Sciences, Purdue University, 1992.

[Gia91] J. C. Giarratano. *CLIPS User's Guide, Version 5.1.* NASA Lyndon B. Johnson Space Center, 1991.

[Gro77] The MATHLAB Group. *MACSYMA Reference Manual, Version 9.* Laboratory for Computer Science, M.I.T., Cambridge, MA, 1977.

[Gro87] Symbolic Computation Group. *Maple User's Manual.* University of Waterloo, Department of Computer Science, Waterloo, Canada, 1987.

[GS91] L. Gross and P. Sternecker. *The Finite Element Tool Package VECFEM.* University of Karlsruhe, 1991.

[Ham62] R. W. Hamming. *Numerical Methods for Scientists and Engineers.* McGraw–Hill, New York, 1962.

[HGJ+89] D. E. Hall, W. H. Greiman, W. F. Johnston, A. X. Merola, S. C. Loken, and D. W. Robertson. The software bus: A vision for scientific software development. *Computer Physics Communications*, 57:211–216, 1989.

[Hib89] Hibbitt, Karlsson & Sorensen, Inc. *ABAQUS User's Manual, Version 4.8*, 1989.

[Hit90] Hitachi, Ltd., Computer Division. *PDEQSOL User's Manual*, 1990.

[HR82] E. N. Houstis and J. R. Rice. High order methods for elliptic partial differential equations with singularities. *International Journal for Numerical Methods in Engineering*, 18:737–754, 1982.

[HR92] E. N. Houstis and J. R. Rice. Parallel ELLPACK: A development and problem solving environment for high performance computing machines. In P. W. Gaffney and E. N. Houstis, editors, *Programming Environments for High–Level Scientific Problem Solving*, pages 229–243. North–Holland, 1992.

[HRC+90] E. N. Houstis, J. R. Rice, N. P. Chrisochoides, H. C. Karathanasis, P. N. Papachiou, M. K. Samartzis, E. A. Vavalis, Ko-Yang Wang, and S. Weerawarana. //ELLPACK: A numerical simulation programming environment for parallel MIMD machines. In J. Sopka, editor, *Proceedings of Supercomputing '90*, pages 96–107. ACM Press, 1990.

[HRCV88] E. N. Houstis, J. R. Rice, C. C. Christara, and E. A. Vavalis. Performance of scientific software. In J. R. Rice, editor, *Mathematical Aspects of Scientific Software*, pages 123–155. Springer–Verlag, 1988.

[HRW94] Elias N. Houstis, John R. Rice, and Sanjiva Weerawarana. An open structure for PDE solving systems. In *Proceedings of the 14th IMACS World Congress on Computation and Applied Mathematics*, 1994. to appear.

[HRWH94] E. N. Houstis, J. R. Rice, S. Weerawarana, and C. E. Houstis. PYTHIA: A computationally intelligent paradigm to support smart problem solving environments for PDE based applications. 1994. to appear.

[IK79] L. Imre and T. Kornyey. *Applied Numerical Methods.* Academic Press, 1979.

[IMS86] IMSL, Inc. *PDE/PROTRAN User's Manual*, 1986.

[JP92] G. O. Cook Jr. and J. F. Painter. ALPAL: A tool to generate simulation codes from natural descriptions. In E. N. Houstis, J. R. Rice, and R. Vichnevetsky, editors, *Expert Systems for Scientific Computing*, pages 401–420. North–Holland, 1992.

[KDMW92] E. Kant, F. Daube, W. MacGregor, and J. Wald. Knowledge–based program generation for mathematical modeling. In E. N. Houstis, J. R. Rice, and R. Vichnevetsky, editors, *Expert Systems for Scientific Computing*, pages 371–392. North–Holland, 1992.

[KLS+94] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steel Jr., and M. E. Zosel. *The High Performance Fortran Handbook.* MIT Press, 1994.

[KME93] M. S. Kamel, K. S. Ma, and W. H. Enright. ODEXPERT: An expert system to select numerical solvers for initial value ode systems. *ACM Transactions on Mathematical Software*, 19(1):44–62, 1993.

[KR68] M. Klerer and J. Reinfelds. *Interactive Systems for Experimental Applied Mathematics.* Academic Press, New York, 1968.

[KU90] S. König and C. Ullrich. An expert system for the economical application of self–validating methods for linear equations. In E. N. Houstis, J. R. Rice, and R. Vichnevetsky, editors, *Intelligent Mathematical Software Systems*, pages 195–220. North–Holland, 1990.

[Mac91a] MacNeal–Schwendler Corporation. *MSC/NASTRAN User's Manual, Volume 1*, 1991.

[Mac91b] G. J. MacRae. Role of high performance computing in environmental modeling. In *Proceedings of Very Large Scale Computations in the 21st Century*, pages 41–72. SIAM, 1991.

[Man90] SunOS Reference Manual. *Network Programming: External Data Representation Standard.* Sun Microsystems, Inc., 1990.

[MOF90] Peter K. Moore, Can Ozturan, and Joseph E. Flaherty. Towards the automatic numerical solution of partial differential equations. In E. N. Houstis, J. R. Rice, and R. Vichnevetsky, editors, *Intelligent Mathematical Software Systems*, pages 15–22. North–Holland, 1990.

[MS79] N. K. Madsen and R. F. Sincovec. PDECOL: General collocation software for partial differential equations. *ACM Transactions on Mathematical Software*, 5(3):326–351, 1979.

[MS81] David K. Melgaard and Richard F. Sincovec. General software for two–dimensional nonlinear partial differential equations. *ACM Transactions on Mathematical Software*, 7(1):106–125, 1981.

[MV88] J. O. Mouton and J. Valusek. E&P software integration: An elusive goal? *World Oil*, pages 48–55, September 1988.

[Nat] National Instruments Corporation, Austin, Texas. *IEEE 488 and VXIbus Control, Data Acquisition, and Analysis*, 1994 edition.

[Nat92] National Instruments Coporation, Austin, Texas. *LabView User's Manual*, 1992.

[OPSS93] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The information bus–an architecture for exensible distributed systems. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 58–68. ACM, 1993.

[Ous90] J. Ousterhout. Tcl: An embeddable command language. In *Proceedings of the USENIX Winter Conference*, pages 133–146, January 1990.

[PAP+89] V. Paxson, C. Aragon, S. Peggs, C. Saltmarsh, and L. Schachinger. A unified approach to building accelerator simulation software for the SSC. In *Proceedings of the 1989 IEEE Particle Accelerator Conference*, volume 1, pages 82–84, 1989.

[Pea88] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann, 1988.

[PS93] Vern Paxson and Chris Saltmarsh. Glish: A user–level software bus for loosely–coupled distributed systems. In *Proceedings of the Winter 1993 Usenix Conference*, pages 217–276. USENIX Association, 1993.

[PSW91] J. Purtilo, R. T. Snodgrass, and A. L. Wolf. Software bus organization: Reference model and comparison of two existing systems. Technical Report TR–8, ARPA Module Interconnection Formalism Working Group, 1991. (file://thumper.cs.umd.edu/files/doc/refmodel.ps.Z).

[Pur94] J. M. Purtilo. The polylith software bus. *ACM Transactions on Programming Languages and Systems*, 16(1):151–174, 1994.

[RB85] J. R. Rice and R. F. Boisvert. *Solving Elliptic Problems Using ELLPACK.* Springer–Verlag, 1985.

[RHD81] John R. Rice, Elias N. Houstis, and Wayne R. Dyksen. A population of linear, second order, elliptic partial differential equations on rectangular domains, part I. *Mathematics of Computation*, 36:475–484, 1981.

[Ric76] J. R. Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976.

[Ric79] J. R. Rice. Methodology for the algorithm selection problem. In L. D. Fosdick, editor, *Performance Evaluation of Numerical Software*, pages 301–307. North–Holland, 1979.

[Ric89] John R. Rice. Composition of libraries, software parts and problem solving environments. Technical Report CSD–TR–852, Department of Computer Sciences, Purdue University, 1989.

[RM86] D. E. Rumelhart and J. L. McClelland. *Parallel Distributed Processing, Explorations into the microstructure of Cognition.* MIT Press, Cambridge, MA, USA, 1986.

[RW89] Th. Ruppelt and G. Wirtz. Automatic transformation of high–level object–oriented specifications into parallel programs. *Parallel Computing*, 10:15–28, 1989.

[SG93] K. Sayre and M. A. Gray. Backtalk: A generalized dynamic communication system for DAI. *Software–Practice and Experience*, 23(9):1043–1057, 1993.

[Sil92] Silicon Graphics, Inc. IRIS Explorer. Technical report, Silicon Graphics, Inc., 1992.

[SSM85] W. Schönauer, E. Schnepf, and M. Müller. *The FIDISOL Program Package.* University of Karlsruhe, Karlsruhe, Germany, 1985.

[Sun91] SunSoft. Open network computing. Technical report, Sun Microsystems Inc., 1991.

[Sun93] SunSoft. *ToolTalk 1.1 User's Guide.* Sun Microsystems, Inc., Mountain View, CA, 1993.

[Swa85] Swanson Analysis Systems, Inc. *ANSYS User's Manual, Version 4.2*, 1985.

[Tho89] Jonathan Thornburg. A PDE compiler for full–metric numerical relativity. In C. R. Evans, L. S. Finn, and D. W. Hobill, editors, *Frontiers in Numerical Relativity*, pages 370–383. Cambridge University Press, 1989.

[TK89] C. G. Takoudis and M. Kastelic. Selective epitaxial growth of silicon in a barrel reactor. *Chem. Engng. Sci.*, 44:2049–2062, 1989.

[UKO92] Y. Umetani, C. Konno, and T. Ohta. Visual PDEQSOL: A visual and interactive environment for numerical simulation. In P. W. Gaffney and E. N. Houstis, editors, *Programming Environments for High–Level Scientific Problem Solving*, pages 259–269. North–Holland, 1992.

[Wan86] Paul S. Wang. FINGER: A symbolic system for automatic generation of numerical programs in finite element analysis. *Journal of Symbolic Computation*, 2:305–316, 1986.

[WCHR92] Sanjiva Weerawarana, Ann C. Catlin, Elias N. Houstis, and John R. Rice. Integrated symbolic–numeric computing in //ELLPACK: Experiences and plans. Technical Report CSD–TR–92–092, Department of Computer Sciences, Purdue University, 1992.

[We94] Sanjiva Weerawarana and etal. The PDESpec language specification. Technical report, Department of Computer Sciences, Purdue University, 1994. to appear.

[WH94] Reginald L. Walker and Elias N. Houstis. A parallel time stepping scheme. Technical report, Department of Computer Sciences, Purdue University, 1994. to appear.

[WHR92] S. Weerawarana, E. N. Houstis, and J. R. Rice. An interactive symbolic-numeric interface to parallel ELLPACK for building general PDE solvers. In Bruce Randall Donald, Deepak Kapur, and Joseph L. Mundy, editors, *Symbolic and Numerical Computation for Artificial Intelligence*, chapter 13, pages 303–322. Academic Press, 1992.

[WHR94a] Sanjiva Weerawarana, Elias N. Houstis, and John R. Rice. A software platform for integrating symbolic computation with a PDE solving environment. In *Proceedings of the 14th IMACS World Congress on Computation and Applied Mathematics*, 1994. to appear.

[WHR+94b] Sanjiva Weerawarana, Elias N. Houstis, John R. Rice, Ann Christine Catlin, Cheryl L. Crabill, Chi Ching Chui, and Shahani Markus. PDELab: An object-oriented framework for building problem solving environments for PDE based applications. In *Proceedings of the Second Annual Object–Oriented Numerics Conference*, pages 79–92, Rogue–Wave Software, Corvallis, OR, 1994.

[Wol88] Stephen Wolfram. *Mathematica: A System for Doing Mathematics by Computer.* Addison–Wesley, 1988.

[WSS92] H. Wietschorke, M. Schmauder, and W. Schönauer. *The CADSOL Program Package.* University of Karlsruhe, Karlsruhe, Germany, 1992.

[WW92] S. Weerawarana and P. S. Wang. A portable code generator for CRAY FORTRAN. *ACM Transactions on Mathematical Software*, 18(3):241–255, 1992.

[ZMH+93] A. Zell, N. Mache, R. Hübner, G. Mamier, M. Vogt, K-U Herrmann, M. Schmalzl, T. Sommer, A. Hatzigeorgiou, S. Döring, and D. Posselt. Stuttgart Neural Network Simulator User Manual, Version 3.1. Technical Report 3, University of Stuttgart, 1993.

Vita

Sanjiva Weerawarana was born in Colombo, Sri Lanka on the fifteenth day of January, 1967. He studied at Thurston College, Colombo (1973–74), Royal College, Colombo (1974–80, 1981–84), the Indian Central School, Muscat, Oman (1980–81), and the American Embassy School, New Delhi, India (1984–85). After receiving a high school diploma from AES, he enrolled at Kent State University, Kent, Ohio in August 1985. He received a B.S. in Applied Mathematics / Computer Science in May 1988 and an M.S. in Applied Mathematics / Computer Science in May 1989. In August 1989, he joined the Department of Computer Sciences at Purdue University and received the degree of Doctor of Philosophy in August 1994. He is a member of ACM, Phi Beta Kappa, Upsilon Pi Epsilon and Phi Kappa Phi.